SAND94-1100
Unlimited Release
Printed April 18, 2000 11:57 am

Distribution
Category UC-705

CUBIT Mesh Generation Environment Volume 1: Users Manual

CUBIT Development Team[1]
Sandia National Laboratories
Albuquerque, New Mexico 87185-0441

Abstract

The CUBIT mesh generation toolkit is a two- and three-dimensional finite element mesh generation tool which is being developed to pursue the goal of robust and unattended mesh generation—*effectively automating the generation of quadrilateral and hexahedral elements*. CUBIT generates surface and volume meshes for solid model-based geometries; these meshes are used for finite element analysis applications. A combination of techniques including paving, mapping, submapping, sweeping, and various other algorithms being developed are available for discretizing the geometry into a finite element mesh. The software is used for both production mesh generation and as a testbed for new algorithms. While CUBIT is specifically designed to reduce the time required to create all-quadrilateral and all-hexahedral meshes, it also provides the capability to generate hex dominant and tetrahedral meshes. This manual is designed to serve as a reference and guide to creating finite element models in the CUBIT environment.

This manual documents CUBIT Version 4.0.

1. See the next page for the members of the CUBIT Development Team.

Cubit Development Team Membership

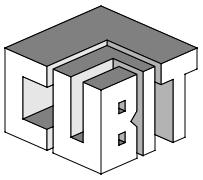
Sandia National Laboratories, Albuquerque New Mexico	
Robert A. Kerr	Parallel Computing Sciences
Patrick Knupp	Parallel Computing Sciences
Robert W. Leland	Manager, Parallel Computing Sciences
Darryl J. Melander	Parallel Computing Sciences
Scott A. Mitchell	Parallel Computing Sciences
Steven J. Owen	Parallel Computing Sciences
Jason F. Shepherd	Parallel Computing Sciences
Timothy J. Tautges	Parallel Computing Sciences
David R. White	Parallel Computing Sciences
Brigham Young University, Provo, Utah	
Steve Benzley	Professor of Civil and Environmental Engineering
Michael J. Borden	Student in Department of Civil and Environmental Eng.
Steven R. Jankovich	Student in Department of Mechanical Engineering
University of Wisconsin	
Jason Kraftcheck	Student in Department of Mechanical Engineering
Yong Lu	Student in Department of Mechanical Engineering
Contractors	
Ray J. Meyers	Contractor, Provo, Utah
MIchael Stephenson	Contractor, Provo, Utah
Caterpillar Inc.	
Steve Storm	Corporate Information Services
Eric Nielsen	Corporate Information Services
Rammagy Yoeu	Corporate Information Services

Table of Contents

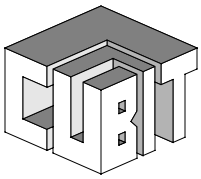
Table of Contents

Table of Contents

List of Figures



List of Tables



Chapter 1: Getting Started

▼ Introduction

Welcome to CUBIT, the Sandia National Laboratory automated mesh generation toolkit. With CUBIT the geometry of a part can be imported, created, and/or modified. The geometry can be discretized into a finite element mesh using a combination of meshing algorithms and boundary conditions can be applied to the mesh through the geometry and appropriate files for analysis generated. CUBIT is designed to reduce the time required to create quadrilateral, triangular, hexahedral, tetrahedral and mixed element meshes, with an emphasis on algorithms and techniques for generating large, unstructured, and high-quality hexahedral meshes.

The CUBIT environment is designed to provide the user with a powerful toolkit of meshing algorithms that require varying degrees of input to produce a complete finite element model. As such, the code is constantly being updated and improved. Feedback from our users indicates that new meshing tools are often needed and/or desired before they have been completely tested and debugged; therefore, the released versin of CUBIT contains algorithms which are to be considered not quite ready for production use. These algorithms are identified in their documentation later in this manual.

Experience has shown that generating meshes for complex, solid model-based geometries requires a variety of tools, from completely automatic tools to tools requiring large amounts of user input. The overall goal of the CUBIT project is to reduce the time to mesh for these problems, and this goal has been achieved by inegrating these tools in a common framework. The user is encouraged to become familiar with the available tools, so that he can choose the right tool for his particular job.

▼ How to Use This Manual

This manual provides specific information about the commands and features of CUBIT. It is divided into chapters which roughly follow the process in which a finite element model is designed, from geometry creation to mesh generation to boundary condition application. An example is provided in a tutorial chapter to illustrate some of the capabilities and uses of

CUBIT. Appendices containing complete command usage, examples, installation instructions, and a list of available colors are included.

Integrated in CUBIT are algorithms and tools which are in a *user beware* state. As they are further tested (often with the assistance of users) and improved, the tool becomes more stable and production-worthy. Since documentation of the tool is necessary for actual use, we have included the documentation of all available tools in the manual. However, to warn the user, a "hammer" icon is placed in the document next to those features which are in a state of work-in-progress (See "hammer" icon in left margin). When using these tools, the user should proceed with caution.

Certain portions of this manual contain information that is vital for understanding and effectively using CUBIT. These portions are hilighted with a "key" icon positioned in the document next to these sections.

This manual documents CUBIT Version 4.0, April 2000.

▼ Features

The CUBIT environment is designed to provide the user with a powerful toolkit of meshing algorithms that require varying degrees of input to produce a complete finite element model. The following sections provide a brief overview of the various features in CUBIT.

Geometry Creation, Modification and Healing

The CUBIT package relies on the ACIS solid modeling engine for geometry representation and querying. Geometry creation is accomplished using the geometric primitives and boolean operations in CUBIT or by reading model from a file in the ACIS SAT file format. SAT files can be written directly from several commercial CAD systems, including SolidWorks and AutoCAD. In addition, geometry models can be generated in and written from other CAD systems and translated to the SAT format; translators are available for many formats, including Pro/Engineer, IGES and STEP. CUBIT can also directly import planar surface geometry in the FASTQ [5] file format, a legacy meshing tool written at Sandia. Finally, there are efforts or plans underway to port CUBIT directly to other CAD systems, including Pro/Engineer and Ideas.

The CUBIT project has purchased a limited number of licenses for geometric healing provided by Spatial Technology. This technology allows the users to "heal" or clean invalid geometric entities and topology resulting from translation or model creation artifacts. Currently, healing is handled by sending "dirty" geometry to members of the CUBIT project for healing. For more information about obtaining a license to do local healing, contact the CUBIT development team.

Non-Manifold Topology

Typical assembly meshes produced using CUBIT require contiguous mesh across multiple parts in an assembly. This "non-manifold topology" is accomplished in CUBIT by representing shared topological surfaces in the geometric model. Geometric models are always imported into CUBIT as manifold models; then, surfaces which are pass a geometric and topological comparison are "merged" to form shared surfaces. A similar technique is used to merge model edges and vertices across parts. These comparisons are performed automatically, and can optionally be restricted to subsets of the model (to allow representations of such features as slide lines).

Geometry Decomposition

Solid models often require decomposition to make them amenable to hexahedral meshing. CUBIT contains a wide variety of tools for interactive geometry decomposition, and a capability for performing automatic geometry decomposition is also under development.

Mesh Generation

CUBIT contains a variety of tools for generating meshes in one, two and three dimensions. While the primary focus of CUBIT is on generating unstructured quadrilateral and hxahedral meshes, algorithms are also available for structured mesh generation and triangle/tetrahedral mesh generation. Several algorithms for generating mixed hex-tet meshes are also being developed.

Boundary Conditions

CUBIT uses the EXODUS-II format specification for exporting mesh data. EXODUS represents boundary conditions on meshes using Element Blocks, Nodesets, and Sidesets. Element Blocks are used to group elements by material type. Nodesets can be used to group nodes for application of nodal boundary conditions, for example enforced displacement or nodal temperature values. Sidesets are used to represent face-based and edge-based boundary conditions like pressure or heat flux.

Using Element Blocks, Nodesets and Sidesets, a mesh and the appropriate boundary conditions can be specified in an analysis-independent manner. Typically this specification is combined with an additional data file which designates the specific type of boundary condition (temperature, displacement, pressure, etc.), along with boundary condition values.

Element Types

Element types supported in CUBIT include 2 and 3 node bars and beams; 4, 8, and 9 node quads; 3, 6, and 7 node triangles, 4, 8, and 9 node shells; 4, 8, 10, and 16 node tetrahedra, 5 node pyramids, and 8, 20, and 27 node hex elements. Element types can be specified before or after mesh generation is performed. Higher order nodes are projected to the solid geometry where appropriate.

Graphics Display Capabilities

CUBIT uses the HOOPS package for its graphics and rendering engine. CUBIT can display geometric and mesh entities in several modes, including hidden line, shaded or wireframe modes. CUBIT supports screen picking of geometric and mesh entities, as well as mouse-controlled operations on the model view like rotate, pan, and zoom. HOOPS contains drivers which take advantage of hardware acceleration on most supported platforms, as well as support for a standard X11 display. PostScript files of any displayed image can also be generated. CUBIT can also be run without graphics, to allow execution in batch mode or over dialup lines.

Command Line Interface

User interaction with CUBIT is performed through a command line interface; no GUI is available at this time (though there are plans for providing a GUI in the near future). Commands can be entered either interactively or in batch mode though a command file. The command line

interface supports the APREPRO command preprocessor, which when combined with CUBIT's scripting capability allows parameterization of CUBIT input.

Hardware Platforms

CUBIT is written in "standard" C++ and is currently supported on Sun Solaris 2.6, Hewlett-Packard (HP-UX 10.20), and Silicon Graphics (IRIX 6.5) unix workstations. CUBIT has also been ported to the Microsoft NT operating system; plans are underway to make this version available to Sandia CUBIT users.

▼ Executing CUBIT

Execution Command Syntax

The command syntax recognized by CUBIT is:

cubit [-help] [-initfile <val>] [-noinitfile] [-solidmodel <val>]
[-batch] [-nographics] [-nojournal] [-journalfile <file>] [-maxjournal <val>]
[-display <val>] [-noecho] [-debug=<val>] [-information={on|off}]
[-warning={on|off}] [-Include <path>] [-fastq <fastq_file>]
[<input_file_list>][<var=value>]...

where the quantities in square brackets **[-options]** are optional parameters that are used to modify the default behavior of CUBIT and the quantities in angle brackets **<values>** are values supplied to the option. Optional arguments to CUBIT are summarized below.

-help Print a short usage summary of the command syntax to the terminal and exit.

-initfile <val> Use the file specified by **<val>** as the initialization file instead of the default initialization file $**HOME/.cubit**.

-noinitfile Do not read any initialization file. The default behavior is to read the initialization file **$HOME/.cubit** or the file specified by the **-initfile** option if it exists.

-solidmodel <val> Read the ACIS solid model geometry information from the file specified by **<val>** prior to prompting for interactive input.

-batch Specify that there will be no interactive input in this execution of CUBIT. CUBIT will terminate after reading the initialization file, the geometry file, and the **<input_file_list>.**

-nographics Run CUBIT without graphics. This is generally used with the **-batch** option or when running CUBIT over a line terminal.

-display Sets the location where the CUBIT graphics system will be displayed, analogous to the DISPLAY environment variable for the X Windows system.

-nojournal Do not create a journal file for this execution of CUBIT. This option performs the same function as the **Journal Off** command. The default behavior is to create a new journal file for every execution of CUBIT.

-journalfile <file> Write the journal entries to **<file>**. The file will be overwritten if it already exists.

-maxjournal <val> Only create a maximum of **<val>** default journal files. Default journal files are of the form **cubit.#.jou** where # is a number in the range 01 to 99.

-noecho Do not echo commands to the console. This option performs the same function as the **Echo Off** command. The default behavior is to echo commands to the console.

-debug=<val> Set to "on" the debug message flags indicated by **<val>**, where **<val>** is a comma-separated list of integers or ranges of integers, e.g. 1,3,8-10.

-information={on|off} Turn on/off the printing of information messages from CUBIT to the console.

-warning={on|off} Turn on/off the printing of warning messages from CUBIT to the console.

-Include=<include_path> Set the patch to search for journal files and other input files to be <**include_path**>. This is useful if you are executing a journal file from another directory and that journal file includes other files that exist in that directory also.

-fastq=<fastq_file> Read the mesh and geometry definition data in the FASTQ file <fastq_file> and interpret the data as FASTQ commands. See Reference [5] for a description of the FASTQ file format.

<input_file_list> Input files to be read and executed by CUBIT. Files are processed in the order listed, and afterwards interactive command input can be entered (unless the -batch option is used.)Read the mesh and geometry definition data in the FASTQ file <fastq_file> and interpret the data as FASTQ commands. See Reference [5] for a description of the FASTQ file format.

<variable=value> APREPRO variable-value pairs to be used in the CUBIT session. Values can be either doubles or character type (character values must be surrounded by double quotes.),

Command options can also be specified using the **CUBIT_OPT** environment variable (See "User Environment Settings" on page 5.)

User Environment Settings

CUBIT can interpret the following environment variables.

DISPLAY: X-Window display to which the graphics window should be displayed (and which screen should be used on displays with multiple monitors).

CUBIT_OPT: Execution command line parameter options. Any valid options described in "Execution Command Syntax" on page 4.

CUBIT_LICENSE: Directory location of MSC Aries tetrahedral mesher license file; by default, this license file is set for the ENGSCI LAN compute server and on the JAL LAN it is located in /var/scrl1/.mscCAERoot on several personal machines. Contact the CUBIT development team for more information on obtaining a license for this mesher.

Initialization File

If the file **$HOME/.cubit** or the file specified by the optional **-initfile <val>** option exists when CUBIT begins executing, it is read prior to beginning interactive command input. This file is typically used to perform initialization commands that do not change from one execution to the next, such as turning off journal file output, specifying default mouse buttons, setting geometric and mesh entity colors, and setting the size of the graphics window.

CUBIT Mailing Lists

A mailing list is used to keep interested users informed of new features, bug-fixes, and other pertinent information about CUBIT. The list can also be used for general discussions with other CUBIT users as well as CUBIT developers. To send questions or comments to this list, send email to cubit@sandia.gov. Users can subscribe to the mailing list by sending a mail message to **majordomo@jal.sandia.gov** with a body consisting of

subscribe cubit

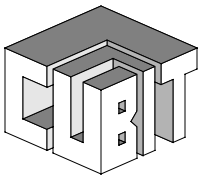
An additional mailing list has been created for direct communication with the CUBIT developers. All messages sent to this list will be distributed to the CUBIT developers only. This list should be used for questions that are not of general interest to other CUBIT users and for reporting bugs in CUBIT. Messages are sent to the CUBIT developers by sending mail to the address:

cubit-dev@sandia.gov

Problem Reports and Enhancement Requests

All CUBIT bugs, problem reports and enhancement requests for CUBIT should be sent to **cubit-dev@sandia.gov**. These requests will be addressed as quickly as possible. The CUBIT developement team will review the problem or enhancement request. Pending the review process, an enhancement request or bug report will be added to CUBIT's bug tracking system, and will be resolved in a timely manner. In general users should expect responses within 48 hours.

Note: The existence and recommended use of an electronic mailing list to report bugs and request enhancements is not intended to discourage face-to-face discussion with CUBIT developers, but rather to minimize response time for bug fixes. Users are encouraged to discuss bugs, enhancements or general meshing issues with the CUBIT development team.



Chapter 2: Tutorial

▼ Introduction

The purpose of this chapter is to demonstrate the capabilities of CUBIT for finite element mesh generation as well as provide a brief tutorial on the use of the software package. This chapter is designed to demonstrate step-by-step instructions on generating a simple mesh on a perforated block.

The following demonstrates the basics of using CUBIT to generate and mesh a geometry. By following this tutorial, you will become familiar with the command-line interface and with as much of the CUBIT environment as possible without stopping for detailed explanations. All the commands introduced in this tutorial are thoroughly documented in subsequent chapters.

Here are a few tips in following the example in the tutorial:

- Focus on instructions preceded with “Step” numbers. These take you through a series of explicit activities that describe exactly what to do to complete the task.

- Refer to screen shots and other pictures that show you what you should see on your own display as you progress through the tutorial.
- An example of the command line is shown below. In this tutorial, the command that you should type will be proceeded by the word “Command” and a colon.

cubit> This is a Command Line

Overview

This tutorial demonstrates the use CUBIT to create and mesh a brick with a through-hole. The primary steps in performing this task are:

- Create geometry
- Set interval sizes and mesh schemes
- Mesh geometry
- Specify boundary conditions
- Export mesh

Each of these steps is described in detail in the following sections.

The geometry for this tutorial is a block with a cylindrical hole in the center, shown in Figure 2-1. This figure also shows the curve and surface identification (ID) numbers, which are

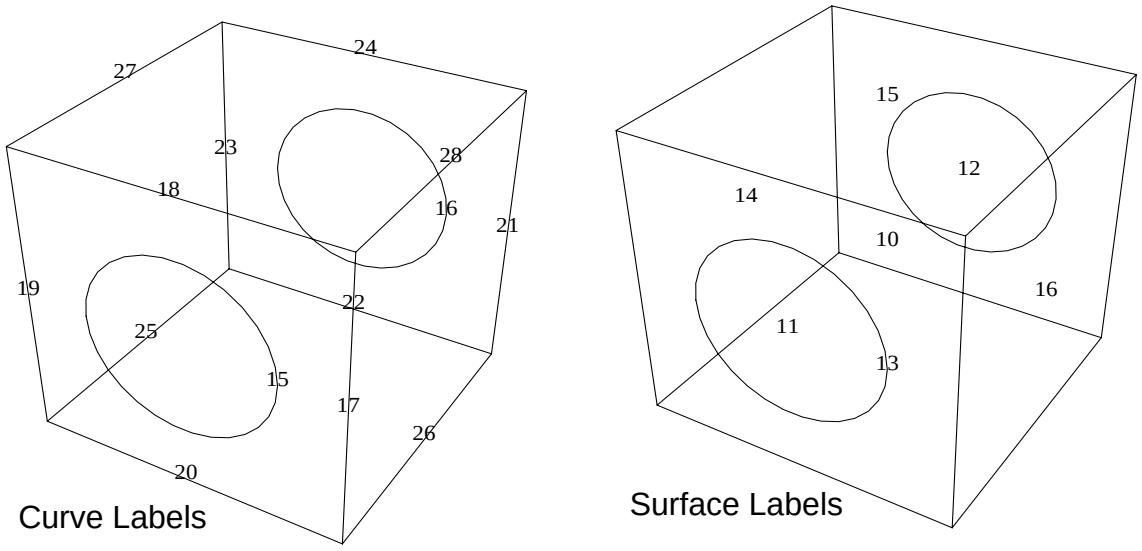


Figure 2-1: Geometry for Cube with Cylindrical Hole

referenced in the command lines shown with each step. The final meshed body is shown in Figure 2-2 and also at the end of this chapter.

Step 1: Beginning Execution

Type “cubit” to begin execution of CUBIT. If you have not yet installed CUBIT, see instructions for doing so in the “CUBIT Installation” Appendix. A CUBIT console window will appear

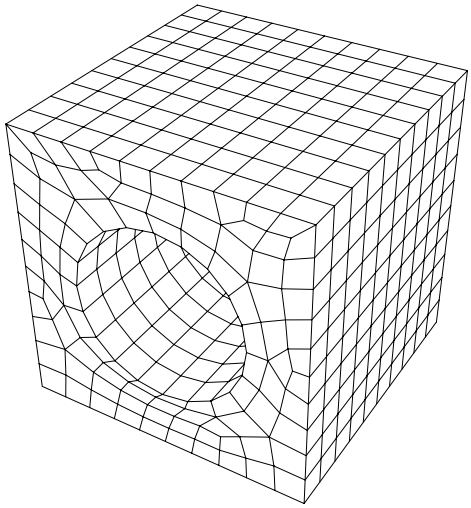


Figure 2-2: Generated Mesh for Cube with Cylindrical Hole

which tells the user which CUBIT version is being run and the most recent revision date. (See

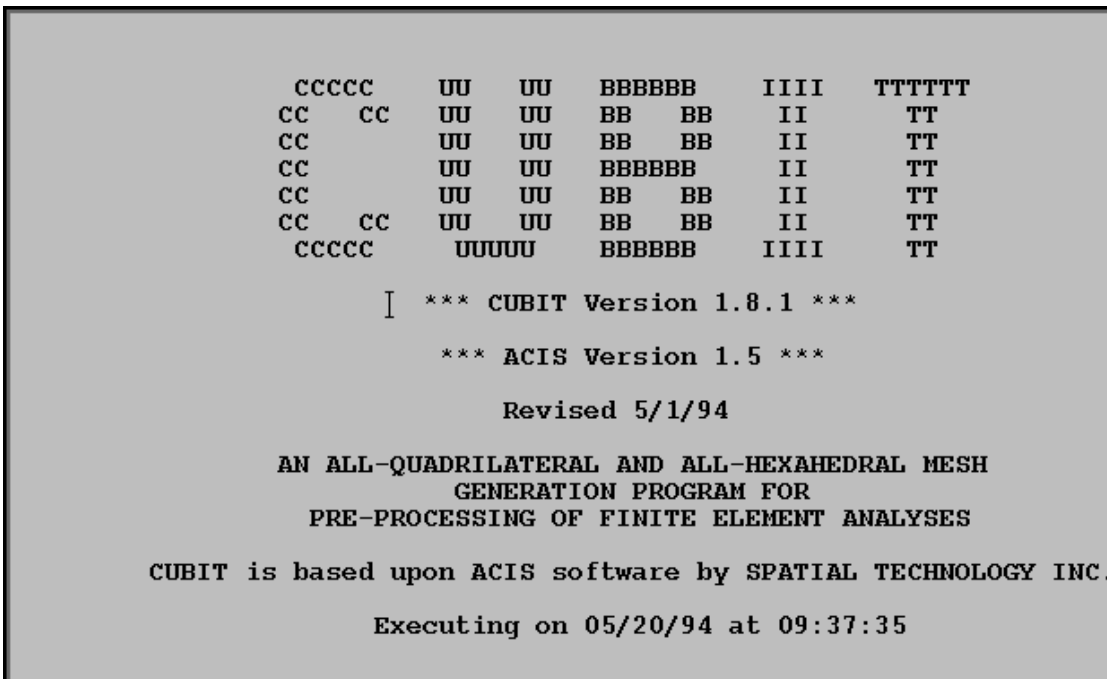


Figure 2-3: CUBIT startup screen.

Figure 2-2 for a picture of this window). This window echos commands and relays information about the success or failure of attempted actions.

Some things to notice are:

- At the bottom of the CUBIT window you will be told where the commands entered in this CUBIT session will be journaled. For example: “Commands will be journaled to ‘cubit01.jou’.
- In addition to the CUBIT version, the code also reports the versions of ACIS and HOOPS that have been compiled into CUBIT (above, versions 1.5 and 2.x, respectively.)

- The command line prompt appears after the banner screen, and appears as “CUBIT>”.
- Commands are entered at that prompt, followed by the “Enter” key.
- Upon startup, a graphics window should also appear, with an axis triad in the lower left hand corner (this window will not appear if CUBIT is started with the -nograpics option.)

Step 2: Creating the Brick

Now you may begin generating the geometry to be meshed. You will create a brick of width 10, depth 10 and height 10. The width and depth correspond to the x and y dimensions of the object being created. The “width” or x-dimension is screen-horizontal and the “depth” or y-dimension is screen-vertical. The height or z-dimension is out of the screen. The command to create this object is:

cubit> Create Brick Width 10. Depth 10. Height 10.

The cube should appear in your display window as shown in Figure 2-4.

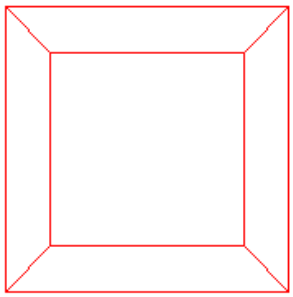


Figure 2-4: Display of brick.

- Note that the journaled version of the command is echoed above the next command line along with the confirmation message “brick body 1 successfully created.”
- The command line is not case-sensitive, so **Brick** and **Width** do not need to be capitalized.
- The “Create” qualifier is optional in this command; also, if the arguments to the Depth and Height qualifiers are identical to that of the Width qualifier, they can be omitted. Therefore, identical results could be achieved with the command “Brick Width 10.”

Step 3: Creating the Cylinder

Now you must form the cylinder which will be used to cut the hole from the brick. This is accomplished with the command

cubit> create cylinder height 12 radius 3

At this point you will see both a cube and a cylinder appear in the CUBIT display window, as shown in Figure 2-5.

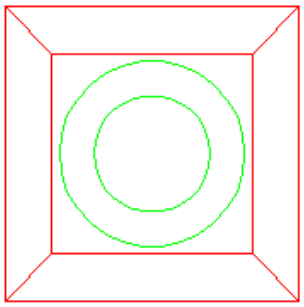


Figure 2-5: Brick and cylinder.

Step 4: Adjusting the Graphics Display

The geometry is drawn in the graphics display in perspective mode, by default from a viewing direction of the +z axis. This view can now be adjusted to verify the proper orientation of the geometry just created. To do this activate your graphics window by placing your cursor in the window or by clicking at the top of it (this will vary depending upon your window settings in your operating system). To change your view, use the Left mouse button to interactively rotate the view, the Middle mouse button to zoom in or out, and the Right mouse button to pan the view. On the GUI version of CUBIT the "control" key must be held down while right, middle or left clicking for transformations. The GUI by default is set up to mouse like other commercial packages so the following button clicks apply: cntrl-right will zoom, cntrl-middle will rotate, and cntrl-left will pan. Graphics changes may also be performed via the command line by specifically setting the view locations (at the command prompt type **help at** or **help from** for the correct syntax).

Use the mouse buttons to make the display look like Figure 2-6.

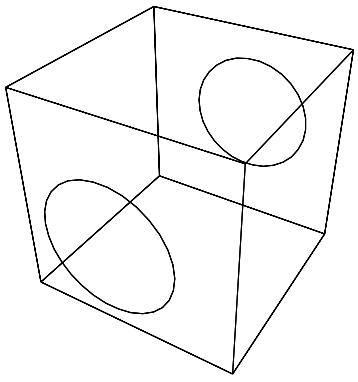


Figure 2-6: View from different perspective.

In the display, the wireframe picture shows the relative locations of the bodies. Viewing the image in shaded mode improves the perspective; this will be described in Step 9: Inspecting the Model.

Step 5: Forming the Hole

Now the cylinder can be subtracted from the brick to form the hole in the block. Issue the following command:

cubit> Subtract 2 From 1

Note: Note that both original bodies are deleted in the boolean operation and replaced with a new body (with an id of 3) which is the result of the boolean operation **Subtract**.

The result of this operation is a single body, a brick with a hole through it. This is shown in Figure 2-7.

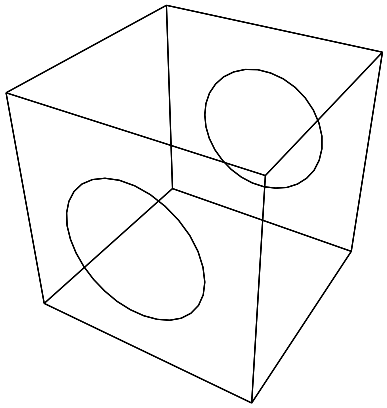


Figure 2-7: Brick after subtracting cylinder.

We have now completed creating the geometry, and are ready to generate a mesh.

Step 6: Setting Interval Sizes

The volume shown in Figure 2-7 will be meshed by sweeping a surface mesh from one side of the block to the other.Before generating any mesh, the user must specify the size of the elements to be generated. In this example, one element size will be specified for the volume as a whole and a smaller size will be specified for around the hole. A direct interval setting will be specified for the sweep direction.

To set the interval size for the overall body, enter the command

cubit> body 3 interval size 1.0

Since the brick is 10 units in length on a side, this specifies that each straight curve is to receive approximately 10 mesh elements.

In order to better resolve the hole in the middle of the top surface, we set a smaller size for the curve bounding this hole. To find the id number of the curve bounding the hole, the user can either pick the curve (See “Selecting Entities with the Mouse” on page 35.) or turn curve labels on and regenerate the view. To do the latter, use the command

cubit> label curve on

cubit> display

The result is shown in Figure 2-8. Then the interval size can be set for the appropriate curve:

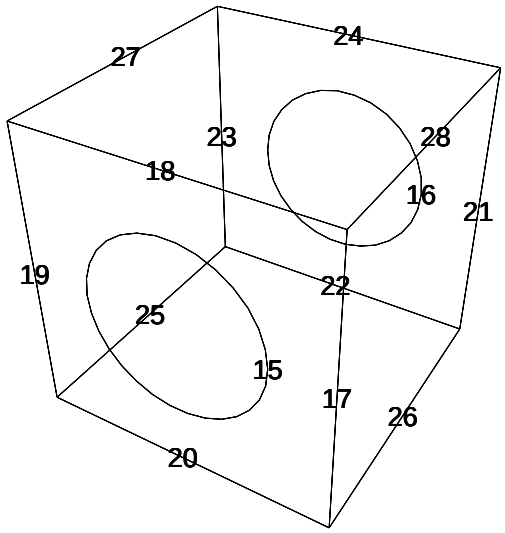


Figure 2-8: Geometry with curve labeling turned on.

cubit> curve 15 interval size.5

Finally, we would like to generate exactly 3 element layers in the sweep direction. This is accomplished by setting the intervals on curve 27:

cubit> curve 27 interval 3

▼ Step 7: Surface Meshing

Now all necessary intervals have been set, and the meshing can proceed. Begin by meshing the front surface (with the hole) using the paving algorithm. This is done in two steps. First set the scheme for that surface to **Pave**, then issue the command to **Mesh**. Since the surface to be paved is number 11, issue the command:[1]

cubit> surface 11 scheme pave

With the meshing scheme specified, we proceed to mesh the surface:

cubit> mesh surface 11

A hidden line view of the result is shown Figure 2-9.

1. The surface id can be obtained using either of the two methods described in the previous step.

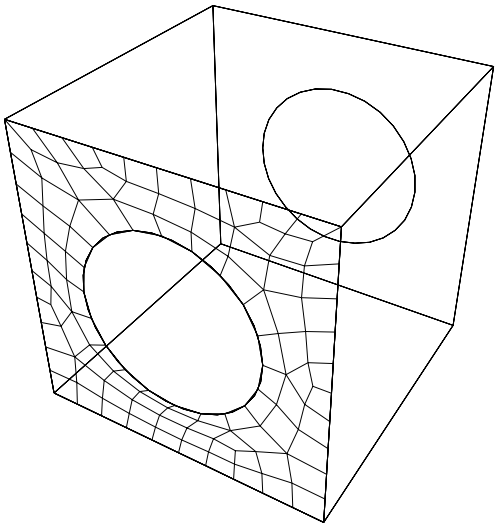


Figure 2-9: Surface meshed with paving.

▼ Step 8: Volume Meshing

The volume mesh can now be generated. Again, the first step is to specify the type of meshing scheme should be used and the second step is to issue the order to mesh. In certain cases, the scheme can be determined by CUBIT automatically. For sweepable volumes, the automatic scheme detection algorithm also identifies the source and target surfaces of the sweep automatically.

To instruct the code to automatically determine the meshing scheme and in this case the source and target surfaces, enter the command

cubit> volume 3 scheme auto

To view the results of auto scheme selection, certain data about the volume can be listed:

cubit> list volume 3

The results of this command are shown in Figure 2-10; note that the scheme, and in this case the source and target surfaces, are reported toward the top of the list output.

With the scheme set, the **mesh** command may be given:

cubit> mesh volume 3

The final meshed body will appear in the display window, as shown in Figure 2-11. By default only the surface mesh is drawn, if you want to see all of the elements you can enter:

cubit> draw hex all in volume 3

```
Volume Entity  (Id = 3)
        Meshed:         No
        Mesh Scheme:    sweep   (automatically selected)
          Source:       Surface 11 (Id=11)
          Target:       Surface 12 (Id=12)
          Sweep Smooth Scheme: Off
        Smooth Scheme:  equipotential fixed

        Interval Count: 1
        Interval Size:  1.000000
        Block Id:       0

       7 Owned Surfaces:        Mesh Scheme                    Interval:
      ______Name______    Id   +is meshed     Smooth Scheme Count   Size
            Surface 10    10      submap-     winslow fixed     1      1
                   Periodic Interval: 3, Soft
            Surface 11    11       pave-      winslow fixed     1      1
            Surface 12    12       pave-      winslow fixed     1      1
            Surface 13    13        map-      winslow fixed     1      1
            Surface 14    14        map-      winslow fixed     1      1
            Surface 15    15        map-      winslow fixed     1      1
            Surface 16    16        map-      winslow fixed     1      1

        In Body 3.
Journaled Command: list volume 3
```

Figure 2-10: Output from listing volume 3

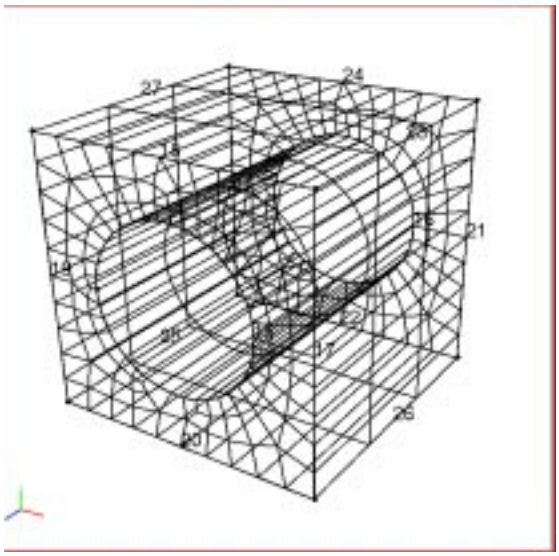


Figure 2-11: Wireframe view of volume mesh.

▼ Step 9: Inspecting the Model

The type, quality, and speed of the rendering the image can be controlled in CUBIT by using several **graphics mode** commands, such as **Wireframe**, **Hiddenline**, and **Smoothshade**. For example:

cubit> graphics mode hiddenline

The hidden line display is illustrated in Figure 2-12. Next, try:

cubit> graphics mode smoothshade

The smoothshade display is also shown Figure 2-12.

For detailed information on the viewing mode options, See “Graphics Modes” on page 29..

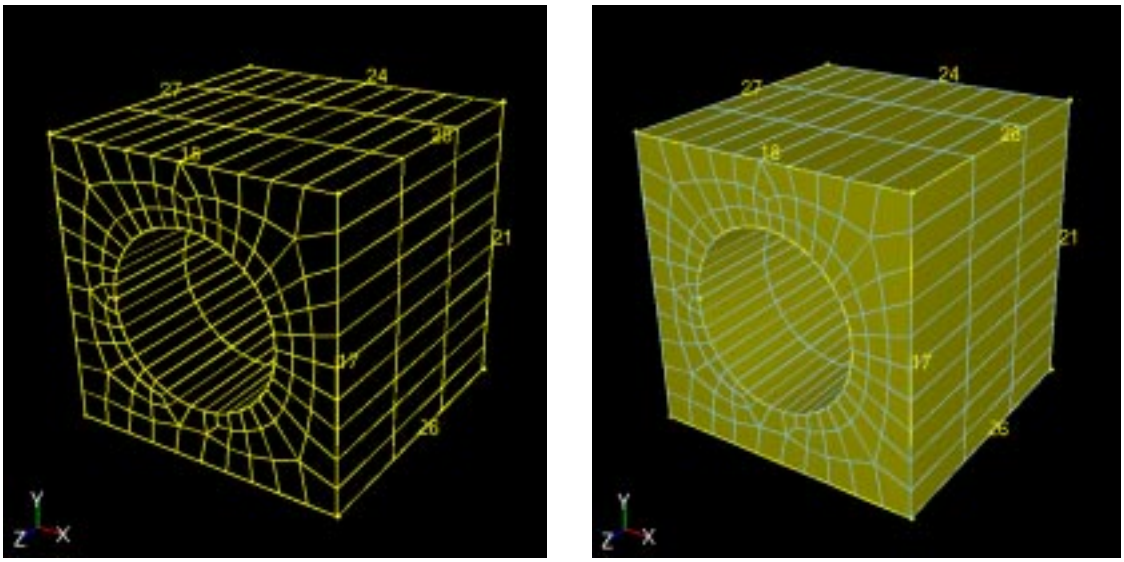


Figure 2-12: Hiddenline (left) and shaded (right) view of volume mesh.

Although CUBIT automatically computes limited quality metrics after generating a mesh and warns the user about certain cases of bad quality, it is still a good idea to inspect a broader set of quality measures. To do this, enter the command

cubit> quality volume 3

The results of the quality output are shown below. For an explanation of each quality metric along with acceptable ranges, see Figure 2-13. For the purposes of this tutorial, you can assume the quality metrics shown below are in an acceptable range.

▼ Step 10: Defining Boundary Conditions

Let us assume that the we need to define one material type for the entire mesh, and a single node-based boundary condition on all surfaces. This accomplished by identifying an Element Block and a Nodeset, respectively; the id numbers assigned to these entities are assigned by the user, usually by some convention meaningful to the analysis to be done. The element block and nodeset are identified using the commands:

Volume 3 Hex quality, 333 elements:

Function Name	Average	Std Dev	Minimum (id)	Maximum (id)
Aspect Ratio	4.887e+00	1.312e+00	2.860e+00 (287)	8.866e+00 (142)
Skew	1.572e-01	1.071e-01	5.640e-03 (332)	4.455e-01 (87)
Taper	1.067e-15	1.054e-15	1.322e-17 (198)	6.916e-15 (223)
Element Volume	2.158e+00	1.089e+00	5.727e-01 (31)	4.593e+00 (176)
Stretch	3.145e-01	8.183e-02	1.557e-01 (148)	4.737e-01 (278)
Diagonal Ratio	9.830e-01	1.647e-02	9.331e-01 (87)	9.994e-01 (221)
Dimension	5.330e-01	1.329e-01	2.868e-01 (31)	7.861e-01 (158)
Oddy	6.200e+01	5.663e+01	1.081e+01 (64)	2.535e+02 (149)
Condition No.	2.711e+00	8.275e-01	1.675e+00 (64)	5.599e+00 (220)
Jacobian	1.728e+00	9.345e-01	4.218e-01 (38)	3.870e+00 (64)
Scaled Jacobian	9.236e-01	6.745e-02	6.467e-01 (109)	9.965e-01 (52)

Journaled Command: quality volume 3

Figure 2-13: Quality table from volume 3's hex mesh

cubit> block 100 volume 3

cubit> nodeset 100 surface all in volume 3

▼ Step 11: Exporting the Mesh

Finally, the mesh needs to be written to an ExodusII file. This is easily done:

cubit> export genesis 'brick_with_hole.g'

The filename and extension are arbitrary and, like the block and nodeset numbers, are usually named according to a convention meaningful to the analysis.

▼ Congratulations

You have created your first CUBIT mesh. The following chapters contain more detailed information about using CUBIT and an in-depth description of the meshing algorithms available.

CHAPTER 2: Tutorial

Chapter 3: Environment

▼ Introduction

The CUBIT user interface is designed to fill multiple meshing needs throughout the analysis process. The user interface options include a traditional command line interface as well as no-graphics and batch mode operation. This chapter covers the interface options as well as the use of journal files, control of the graphics, a description of methods for obtaining model information, and an overview of the help facility.

▼ Command Syntax

The execution of CUBIT is controlled either by entering commands from the command line or by reading them in from a journal file. Throughout this document, each function or process will have a description of the corresponding CUBIT command; in this section, general conventions for command syntax will be described. The user can obtain a quick guide to proper command format by issuing the **<keyword> help** command; see “Obtaining Help” on page 58 for details.

CUBIT commands are described in this manual and in the help output using the following conventions. An example of a typical CUBIT command is:

{volume_list} Scheme Project [Source {surface_list} Target {surface_list}]

The commands recognized by CUBIT are free-format and abide by the following syntax conventions.

- Case is not significant.
- The "**#**" character in any command line begins a comment. The "**#**" and any characters following it on the same line are ignored.
- Commands may be abbreviated as long as enough characters are used to distinguish it from other commands.
- The meaning and type of command parameters depend on the keyword. Some parameters used in CUBIT commands are:
- **Numeric:** A numeric parameter may be a real number or an integer. A real number may be in any legal C or FORTRAN numeric format (for example, 1, 0.2, -1e-2). An integer parameter may be in any legal decimal integer format (for example, 1, 100, 1000, but not 1.5, 1.0, 0x1F).
- **String:** A string parameter is a literal character string contained within single or double quotes. For example, **'This is a string'**.
- **Filename:** A filename parameter must specify a legal filename on the system on which CUBIT is running. The filename must be specified using either a relative path (**../cubit/mesh.jou**), a fully-qualified path (**/home/jdoe/cubit/mesh.jou**), or no path; in the latter case, the file must be in the working directory or in a directory specified using the -path option to CUBIT (see "Executing CUBIT" on page 4 for details.) Like a string, the file name must be contained within single or double quotes.Environment variables and aliases may not be used in the filename specification; for example, the C-Shell shorthand of referring to a file relative to the user's login directory (**~jdoe/cubit/mesh.jou**) is not valid.
- **Toggle:** Some commands require a "toggle" keyword to enable or disable a setting or option. Valid toggle keywords are "**on**", "**yes**", and "**true**" to enable the option; and "**off**", "**no**", and "**false**" to disable the option.
- Each command typically has either:
- an action keyword or "verb" followed by a variable number of parameters, for example

  Mesh Volume 1

  Here **Mesh** is the verb and **Volume 1** is the parameter.
- or a selector keyword or "noun" followed by a name and value of an attribute of the entity indicated, for example

  Volume 1 Scheme Project Source 1 Target 2

  Here **Volume 1** is the noun, **Scheme** is the attribute, and the remaining data are parameters to the **Scheme** keyword.

The notation conventions used in the command descriptions in this document are:

- The command will be shown in a format that **looks like this**,
- A word enclosed in angle brackets (**<parameter>**) signifies a user-specified parameter. The value can be an integer, a range of integers, a real number, a string, or a string denoting a

filename or toggle. The valid value types should be evident from the command or the command description.

- A series of words delimited by a vertical bar (**choice1 | choice2 | choice3**) signifies a choice between the parameters listed.
- A word enclosed in square brackets (**[optional]**) signifies optional input which can be entered to modify the default behavior of the command.

▼ Executing CUBIT

Execution Command Syntax

The command syntax recognized by CUBIT is:

cubit [-help] [-initfile <val>] [-noinitfile] [-solidmodel <val>] [-batch] [-nographics] [-nojournal] [-journalfile <file>] [-maxjournal <val>] [-display <val>] [-noecho] [-debug=<val>] [-information={on|off}] [-warning={on|off}] [-Include <path>] [-fastq <fastq_file>] [<input_file_list>][<var=value>]...

where the quantities in square brackets **[-options]** are optional parameters that are used to modify the default behavior of CUBIT and the quantities in angle brackets **<values>** are values supplied to the option. Optional arguments to CUBIT are summarized below.

-help Print a short usage summary of the command syntax to the terminal and exit.

-initfile <val> Use the file specified by **<val>** as the initialization file instead of the default initialization file $**HOME/.cubit**.

-noinitfile Do not read any initialization file. The default behavior is to read the initialization file **$HOME/.cubit** or the file specified by the **-initfile** option if it exists.

-solidmodel <val> Read the ACIS solid model geometry information from the file specified by **<val>** prior to prompting for interactive input.

-batch Specify that there will be no interactive input in this execution of CUBIT. CUBIT will terminate after reading the initialization file, the geometry file, and the **<input_file_list>.**

-nographics Run CUBIT without graphics. This is generally used with the **-batch** option or when running CUBIT over a line terminal.

-display Sets the location where the CUBIT graphics system will be displayed, analogous to the DISPLAY environment variable for the X Windows system.

-driver <driver_type> Sets the type of graphics display driver to be used. Available drivers depend on platform, hardware, and system installation. Typical drivers include X11 and OpenGL.

-nojournal Do not create a journal file for this execution of CUBIT. This option performs the same function as the **Journal Off** command. The default behavior is to create a new journal file for every execution of CUBIT.

-journalfile <file> Write the journal entries to **<file>**. The file will be overwritten if it already exists.

-maxjournal <val> Only create a maximum of **<val>** default journal files. Default journal files are of the form **cubit.#.jou** where # is a number in the range 01 to 99.

-noecho Do not echo commands to the console. This option performs the same function as the **Echo Off** command. The default behavior is to echo commands to the console.

-debug=<val> Set to “on” the debug message flags indicated by **<val>**, where **<val>** is a comma-separated list of integers or ranges of integers, e.g. 1,3,8-10.

-information={on|off} Turn on/off the printing of information messages from CUBIT to the console.

-warning={on|off} Turn on/off the printing of warning messages from CUBIT to the console.

-Include=<include_path> Set the patch to search for journal files and other input files to be <**include_path**>. This is useful if you are executing a journal file from another directory and that journal file includes other files that exist in that directory also.

-fastq=<fastq_file> Read the mesh and geometry definition data in the FASTQ file <fastq_file> and interpret the data as FASTQ commands. See Reference [5] for a description of the FASTQ file format.

<variable=value> APREPRO variable-value pairs to be used in the CUBIT session. Values can be either doubles or character type (character values must be surrounded by double quotes.),

Command options can also be specified using the **CUBIT_OPT** environment variable (See “User Environment Settings” on page 5.)

<input_file_list> Input files to be read and executed by CUBIT. Files are processed in the order listed, and afterwards interactive command input can be entered (unless the -batch option is used.)Read the mesh and geometry definition data in the FASTQ file <fastq_file> and interpret the data as FASTQ commands. See Reference [5] for a description of the FASTQ file format.

Environment Variables

CUBIT uses the following environment variables.

DISPLAY: X-Window display to which the graphics window should be displayed (and which screen should be used on displays with multiple monitors).

CUBIT_OPT: Execution command line parameter options. Any valid options described in “Execution Command Syntax” on page 21.

CUBIT_LICENSE: Directory location of MSC Aries tetrahedral mesher license file; by default, this license file is located in /var/scrl1/.mscCAERoot on 836 and 880 LAN compute servers. Contact the CUBIT development team for more information on obtaining a license for this module.

HOOPS_PICTURE: Sets the driver type and display to be used by the graphics system. Takes precedence over the DISPLAY environment variable. The format is **driver_type/machine_name:display**. An example is **opengl/mycomputer:0.0**.

Initialization File

If the file **$HOME/.cubit** or the file specified by the **-initfile <val>** option exists when CUBIT begins executing, it is read prior to beginning interactive command input. This file is typically used to perform initialization commands that do not change from one execution to the next, such as turning off journal file output, specifying default mouse buttons, setting geometric and mesh entity colors, and setting the size of the graphics window.

Session Control

The following commands are used to control CUBIT execution.

- **Exit**: The CUBIT session can be discontinued with either of the following commands

  Exit

  Quit

- **Reset**: A reset of CUBIT will clear the CUBIT database of the current geometry and mesh model, allowing the user to begin a new session without exiting CUBIT. This is accomplished with the command

  Reset [Genesis | Blocks | Nodesets | Sidesets]

  A subset of portions of the CUBIT database to be reset can be designated using the qualifiers listed. Advanced options controlled with the **Set** command are not reset.

- **Version:** To determine information on version numbers, enter the command **Version**.. This command reports the CUBIT version number, the date and time the executable was compiled, and the version numbers of the ACIS solid modeler and the HOOPS library linked into the executable. This information is useful when discussing available capabilities or software problems with CUBIT developers.

- **Command echo:** By default, commands entered by the user will be echoed to the terminal. The echo of commands is controlled with the command:

  [set] echo {on | off}

Command Recording and Playback

Sequences of CUBIT commands can be recorded and used as a means to control CUBIT from ASCII text files. Command or “journal” files can be created within CUBIT, or can be created and edited directly by the user outside CUBIT.

Journal File Creation & Playback

Command sequences can be written to a text file, either directly from CUBIT or using a text editor. CUBIT commands can be read directly from a file at any time during CUBIT execution, or can be used to run CUBIT in batch mode. To begin and end writing commands to a file from within CUBIT, use the command

Record ’<filename>’

Record Stop

Once initiated, all commands are copied to this file after their successful execution in CUBIT.

To replay a journal file, issue the command

Playback ’<filename>’

Journal files are most commonly created by recording commands from an interactive CUBIT session, but can also be created using automatic journalling (see below) or even by editing an ASCII text file. Commands being read from a file can represent either the entire set of

commands for a particular session, or can represent a subset of commands the user wishes to execute repeatedly.

Two other commands are useful for controlling playback of CUBIT commands from journal files. Playback from a journal file can be terminated by placing the **Stop** command after the last command to be executed; this causes CUBIT to stop reading commands from the current journal file. Playback can be paused using the **Pause** command; the user is prompted to hit a key, after which playback is resumed.

Journal files are most useful for running CUBIT in batch mode, often in combination with the parameterization available through the Aprepros capability in CUBIT (see ...). Journal files are also useful when a new finite element model is being built, by saving a set of initialization commands then iteratively testing different meshing strategies after playing that initialization file.

Automatic Journal File Creation

By default, CUBIT automatically creates a journal file each time it is executed. The file is created in the current directory, and its name begins with the word "`cubit`" followed by a number between 01 and 99[1], e.g. `cubit01.jou` . Journal file names end with a ".jou" extension, though this is not strictly required for user-generated journal files. If no journalling is desired, the user may start CUBIT with the **-nojournal** command line option or use the command**:**

[set] Journal {Off | On}

Turning journalling back on resumes writing commands to the same journal file[2].

Most CUBIT commands entered during a session are journalled; the exceptions are commands that require interactive input (such as **Zoom Cursor**), some graphics related commands, and the **Play** command. All graphics related commands may be enabled or disabled with the command:

Journal Graphics {On | Off}

The default is **Journal Graphics Off**.

When an entity is specified in a command using its name, the command may be journalled using the entity name, or by using the corresponding entity type and id. The method used to journal commands using names is determined with the command:

Journal Names {On | Off}

The default is **Journal Names On**.

Note: If an entity is referred to using its entity type and id, the command will be journalled with the entity type and id, even if the entity has been named.

---

1. This number increments for each new journal file generated in that directory
2. If CUBIT is started with the **-nojournal** option, journalling cannot be resumed with the **Journal On** command.

Restart

CUBIT has a limited restart capability, which can be used to recover the geometry and mesh saved in a previous session. A session is saved and restored using the following commands:

Save [Restart] [[File] 'filename'] [<entity_list>]
Restore [Restart] [[File] 'filename']

If a file name is entered, the geometry is saved to or restored from that file. On the save command, if an entity list is given, only those geometry entities are saved in the geometry file.

CUBIT uses geometry attributes to embed restart information in the geometry file; by default, the file used to store geometry is named "cubit_geom.sat". If mesh is present when Save is entered, this data is stored in another file, named "cubit_mesh.g". Care should be taken not to overwrite any restart files the user wants to save beyond the next invocation of Save.

Data that are not saved when Save is used include global settings data (debug flags, persistent ids, etc.) and graphics options (shading mode, background color, etc.). These data will be incorporated into the restart capability in a future release.

In addition to saving all possible attributes with the geometry, users have the option of turning on or off individual attributes, e.g. entity ids, groups, etc. See (section on attributes in chapter 4) for more details.

Entity Specification

CUBIT identifies objects in the geometry, mesh, and elsewhere using ID numbers and sometimes names. IDs and names are used in most commands to specify which objects on

which the command is to operate. These objects can be specified in CUBIT commands in a variety of ways, which are best introduced with the following examples (entity range input is italicized):

- **General ranges:** *Surface 1 2 4 to 6 by 2 3 4 5* Scheme Pave
- **Combined geometry, mesh, and genesis entities:** Draw *Sideset 1 Curve 3 Hex 2 4 6*
- **Geometric topology traversal:** *Vertex in Volume 2* Size 0.3
- **Mesh topology traversal:** Draw *Edge in Hex 32*
- ***All* keyword:** List *Block all*
- ***Expand* keyword:** *my_curve_group expand* Scheme Bias Factor 1.5
- ***Except* keyword:** List *Curve 1 to 50 except 2 4 6*

Types of Entity Range Input

The types of entity range input available in CUBIT can be classified in 4 groups:

• General range parsing

Entity ids can be listed in ranges using multiple combinations of id lists (3 4 5 ...) and/or id ranges (1 to 7 by 2). In addition, the “all” identifier can be used anywhere a range is input. For example:

Draw Surface 1 2 4 to 6 Vertex all

• Topological traversal

Topological traversal is indicated using the “in” identifier, can span multiple levels in a heirarchy, and can go either up or down the topology tree. For example, the following entity lists are all valid:

- Vertex in Volume 3
- Volume in Vertex 2 4 6
- Curve 1 to 3 in Body 4 to 8 by 2

If ranges of entities are given on both sides of the “in” identifier, the intersection of the two sets results. For example, in the last command above, the curves that have ids of 1, 2 or 3 and are also in bodies 4, 6 and 8 are used in the command.

At this time, topology traversal is valid only within a particular entity type (mesh entities or geometry entities) and not across entity types; no traversals are provided for genesis entities. Therefore, the following entity lists would not be valid:

- Node in Surface 3 (invalid!)
- Surface in Edge 362 (invalid!)
- Surface in Nodeset 3 (invalid!)

• Exclusion

Entity lists can be entered then filtered using the “except” identifier. This identifier and the ids following it apply only to the immediately preceeding entity list, and are taken to be the same entity type. For example, the following entity lists are valid:

- Curve all except 2 4 6

- Curve 1 2 5 to 50 except 2 3 4
- Curve all except 2 3 4 in surface 2 to 10
- Curve in surface 3 except 2 (produces empty entity list!)

• Group expansion

Groups in CUBIT can consist of any number of geometry entities, and the entities can be of different type (vertex, curve, etc.). Operations on groups can be classified as operations on the group itself or operations on all entities in the group. If a group identifier in a command is followed immediately by the ‘expand’ qualifier, the contents of the group(s) are substituted in place of the group identifier(s); otherwise the command is interpreted as an operation on the group as a whole. If a group preceding the ‘expand’ qualifier includes other groups, all groups are expanded in a recursive fashion.

For example, consider group 1, which consists of surfaces 1, 2 and curve 1. Surfaces 1 and 2 are bounded by curves 2, 3, 4 and 5. The commands in Table 3-1 illustrate the behavior of the ‘expand’ qualifier.

Table 3-1: Parsing of group commands; Group 1 consists of Surfaces 1-2 and Curve 1; Surfaces 1 and 2 are bounded by Curves 2-5.

Command	Entity list produced
`Curve in group 1`	Curve 1
`Curve in group 1 expand`	Curves 1, 2, 3, 4, 5

The ‘expand’ qualifier can be used anywhere a group command is used in an entity list; of course, commands which apply only to groups will be meaningless if the group id is followed by the ‘expand’ qualifier.

Precedence of “Except” and “In”

Several keywords take precedence over others, much the same as some operators have greater precedence in coding languages. In the current implementation, the keyword “Except” takes precedence over other keywords, and serves to separate the identifier list into two sections. Any identifiers following the “Except” keyword apply to the list of entities excluded from the entities preceding the “Except”. Table 3-2 shows the entity lists resulting from selected commands.

Table 3-2: Precedence of “Except” and “In” keywords; Group 1 consists of Surfaces 1-2 and Curve 1.

Command	Entity list produced
`Curve all except 1 in Group 1`	(All curves except curve 1)
`Curve all except 2 3 4 in Surf 2 to 10`	(All curves except 2, 3, 4)

In the first command, the entities to be excluded are the contents of the list “[Curve] 1 in Group 1”, that is the intersection of the lists “Curve 1” and “Curve in Group 1”; since the only curve in Group 1 is Curve 1, the excluded list consists of only Curve 1. The remaining list, after removing the excluded list, is all curves except Curve 1.

In the second command, the excluded list consists of the intersection of the lists “Curve 2 3 4” and “Curve in Surf 2 to 10”; this intersection turns out to be just Curves 2, 3 and 4. The remaining list is all curves except those in the excluded list.

Placement in CUBIT Commands

In general, anywhere a range of entities is allowed, the new parsing capability can be used. However, there can be exceptions to this general rule, because of ambiguities this syntax would produce. Currently, the only exception to this rule is the command used to define a sideset for a surface with respect to an owning volume.

▼ Command Line Editing

The CUBIT command line interface supports an EMACS-style line editing input package for entering commands[1]. It allows the user to edit the current line and move through a list of previous commands. Commands replayed from a journal file are not saved in the history list. The keys used for command line editing are defined in Table 3-3.

Table 3-3: Command Line Interface Line Editing Keys

Key[a]	Function
^A, ^E	Move to beginning or end of line, respectively
^F, ^B	Move forward or backward one position in the current line.
^D	Delete the character under the cursor. Sends end-of-file if no characters on the current line.
^H, DEL[b]	Delete the character to the left of the cursor.
^K	Delete from the current cursor position to the end of the line
^P, ^N	Move to the previous or next line in the history buffer.
^L	Redraw the current line.
^U	Delete the entire line.
NL, CR[c]	Places current input on the history list, appends a newline and returns that line to the CUBIT program for parsing.
?	Provides “instant” help; see “Obtaining Help” on page 58 for details.

1. The command line interface package used in CUBIT is Copyright 1991 by Chris Thewalt. The following copyright notice appears in the source code: “Permission to use, copy, modify, and distribute this software for any purpose and without fee is hereby granted, provided that the above copyright notices appear in all copies and that both the copyright notice and this permission notice appear in supporting documentation. This software is provided “as is” without express or implied warranty”.

a. The notation ^X refers to holding down the control key and then typing the letter X. Case is not significant.
b. See the documentation for your keyboard/workstation to determine which key sends the DEL character.
c. NL is a newline, typically ^J, CR is a carriage return entered the normal way you end a line of text.

Graphics

The graphics display windows present a graphical representation of the geometry and/or the mesh. The quality and speed of rendering the graphics, the visibility, location and orientation of objects in the window, and the labeling of entities, among other things, can all be controlled by the user.

Unless the **-nographics** option was entered on the command line, a graphics window with a black background and an axis triad will appear when CUBIT is first launched. The geometry and mesh will appear in this window, and can be viewed from various camera positions and drawn in various modes (wireframe, hiddenline, shaded, etc.). This section will discuss methods for manipulating the graphics with the mouse and for controlling the appearance of entities drawn in the graphics window.

Graphics in CUBIT operates on the principle of a “display list”, which keeps track of various entities known to the graphics. All geometry and mesh objects created in CUBIT are put into the display list automatically. The visibility and various other attributes of entities in the display list can be controlled individually. In addition, CUBIT can also optionally display entities in an temporary mode, independent of their visibility in the display list. Drawing of items in temporary mode can be combined with the display list to customize the appearance. The overall display is controlled by various attributes like graphics mode, camera position, andlighting, to further enhance the graphics functionality.

Updating the Display

Among the most common graphics-related commands is **Display**. This command clears all highlighting and temporary drawing, and then redraws the model according to the current graphics settings. Two related commands are **Graphics Flush**, which redraws the graphics without clearing highlighting or temporary drawing, and **Graphics Clear**, which clears the graphics window without redrawing the scene, leaving the window blank.

Note: Although most changes to the model are immediately reflected in the graphics display, some are not (for graphics efficiency). Typing **Display** will update the display after such commands.

Graphics Modes

By default, the scene is viewed as a wireframe model. That is, only curves and edges are drawn, and surfaces are transparent. Surfaces can be drawn differently by changing the graphics mode:

Graphics Mode {Wireframe | Hiddenline | Smoothshade | Truehiddenline | Flatshade | Polygonfill | Painters}

The first three modes listed above are used most often; a sample geometry and mesh displayed in each of these modes is shown in Figure 3-1. These modes have the following distinguishing

Figure 3-1: Examples of three most common viewing modes in CUBIT; Wireframe (left); Hiddenline (center); Smoothshade (right).

characteristics:

- **WireFrame** - Surfaces are transparent. This is the fastest graphics mode.
- **HiddenLine** - Surfaces are not drawn, but they obscure what is behind them, giving a more realistic representation of the view.
- **SmoothShade** - Surfaces are filled and shaded. Shaded colors are interpolated across the entire surface. This is the slowest graphics mode, but produces the most realistic results.

The remaining modes are useful in specific circumstances, can be used to refine the appearance of the display, or result in a display of slightly lower quality but one which is generated more quickly. Examles of each are shown in Figure 3-2; their distinguishing characteristics are:

Figure 3-2: Examples of other viewing modes in CUBIT; Flatshade (top left); Polygonfill(top right); Painters(bottom left); Truehiddenline (bottom right).

- **FlatShade** - Similar to Smoothshade, but with each facet of the surface drawn in a constant instead of interpolated color. Gives slightly poorer display quality but with increase speed compared to Smoothshade.
- **PolygonFill** - Surfaces are filled but not shaded. This is a relatively fast mode, but has relatively poor quality.
- **Painters** - Similar to Smoothshade mode, but *may* render more quickly, albeit with poorer quality. This mode is slightly slower than FlatShade mode on most machines.
- **Transparent** - Renders surfaces as semi-transparent shaded images, allowing objects to shine-through from behind. Is not supported on all platforms, and generally requires advanced graphics hardware.
- **Truehiddenline** - Similar to Hiddenline mode, but gives better results with a slight speed penalty. TrueHiddenLine mode also gives you additional options:

  Graphics TrueHiddenLine Visibiltiy {on|off} - If this option is turned off, TrueHiddenLine mode looks the same as HiddenLine mode. If it is turned on, geometry that falls behind a surface is dimmed instead of invisible.

  Graphics TrueHiddenLine Dim Factor <factor> - This determines how dim the lines behind surfaces are drawn. Factor may range from 0 (invisible) to 1 (full brightness).

  Graphics TrueHiddenLine Pattern <pattern> - This determines what pattern is used to draw lines behind surfaces (e.g. dotted, dashed, etc.; see online help for a list of valid line patterns).

There is another option that is similar to a graphics mode, set with the command

Graphics Use Facets [On|Off]

This command determines how shaded and filled surfaces are drawn when they are meshed. If Graphics Use Facets is on, the mesh facets (element faces) are used to shade the model. This is

particularly helpful for curved surfaces which may cut through some of the mesh faces; see Figure 3-3.

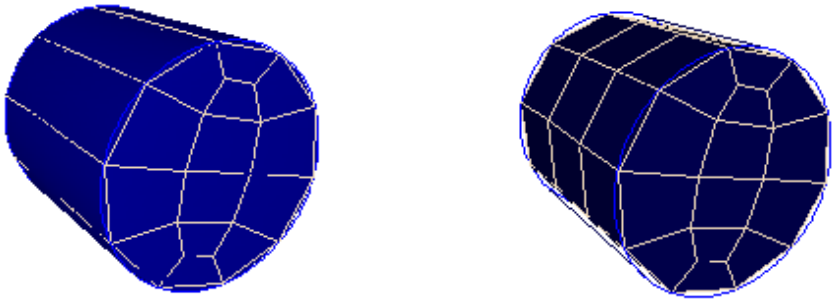


Figure 3-3: A meshed cylinder shown with graphics facets off (left) and graphics facets on (right); note how geometry facets on the curved surface obscure mesh edges when facets are off.

Drawing and Highlighting Entities

In order to effectively visualize the model, it is often necessary to draw an entity by itself, or several entities as a group. This is easily done with the command

Draw {Entity specification} [Add]

where Entity specification is an entity list as descibed in “Entity Specification” on page 25. This command clears the display before drawing the specified entity or entities. If the **Add** option is specified, the given entity is added to what is already drawn on the screen. The entities specified in this command are drawn regardless of their visibility setting (see “Geometry and Mesh Entity Visibility” on page 41 for more details about visibility).

Entities can be highlighted using the command

Highlight {Entity specification}

This command highlights the specified entities in the current display with the current highlight color. Highlighting can be removed using the command

Graphics Clear Highlight

To return to the normal display of the entire model, type **Display**.

Drawing Other Objects

In addition to the common geometry, mesh and genesis entities, other objects may be drawn with variations of the Draw command. As with the other Draw commands, typing **Display** after drawing these objects will restore the scene to its normal display.

- **Entity Normals**

  The normal to a surfaces, face, or tri may be drawn with the command

  Draw {Surface | Face | Tri} <id_range> Normal [Length <length>] [Face | Tri]

  The normal is drawn as a line of length **length** (a length of 1 is the default), starting at the centroid of the entity, pointing in the direction of the entity’s normal. If the normal is being

drawn for a surface, the normal for all faces or tris that belong to a surface may be drawn by using the **Face** or **Tri** qualifier with the command.

- **Volume Sources and Targets**

  Once the source and target surfaces have been set on a volume that will be meshed with the sweep algorithm, the source and target may be visually identified with the command

  Draw Volume <volume_id_range> [Source][Target] [Length <size>]

  If the **Source** keyword is included, the normal of the source surface or surfaces will be drawn in green into the specified volume. If the **Target** keyword is included, the normal of the target surface or surfaces will be drawn in red into the specified volume.

- **Model Axis**

  The model axis may be drawn with the command

  Draw Axis [Length <length>]

  The axis is drawn as three lines beginning at the model origin, one line in each of the three coordinate directions. The length of those lines is determined by the **length** parameter, which defaults to 1.

Mouse-Based View Navigation

The mouse can be used to navigate through the scene using various view transformations. These transformations are accomplished by clicking a mouse button in the graphics window and dragging, sometimes while holding a modifier key such as Shift or Control. When run with graphics on, CUBIT is always in mouse mode; that is, mouse-based transformations are always available, without needing to enter a CUBIT command.

Mouse-based view transformations are accomplished by placing the pointer in the graphics window and then either holding down a mouse button and dragging, or by clicking on a location in the graphics window. Some functions also require one or more modifier keys to be held down; the modifier keys used in CUBIT are Shift (Sh), Control (Ctl), and Alt. Each of the available view transformations has a default binding to a mouse button-modifier key combination; this binding can be changed by the user if desired. Transformations and button mappings are summarized in Table 3-4.

Table 3-4: Default Mouse Function Mappings

Function	Description	Binding
Rotate	Rotates the scene about the camera axis. Dragging the mouse near the center of the graphics window will rotate about the camera's X- or Y-axis; dragging the mouse near the edge of the window will rotate about the Z-axis (i.e. about the camera's line of sight). Type a **u** in the graphics window to see the dividing line between the two types of rotation.	B1
Zoom	Zooms the scene in or out by clicking the mouse in the graphics window and dragging up or down.	B2
Pan	"Drags" the scene around with the mouse.	B3
Rotate XY	Rotates the scene about the X- or Y- camera axis but not about the Z-axis.	Alt-B1
Rotate Z	Rotates the scene about the Z- camera axis (the camera's line of sight).	Sh-Alt-B1

Table 3-4: Default Mouse Function Mappings

Function	Description	Binding
Navigational Zoom	Zooms the scene by moving both the camera and its focal point forward.	Sh-B2
Telephoto Zoom	Zooms the scene by decreasing the field of view.	Sh-Ctl-B2
Zoom Cursor (Click)	Zooms the scene after user clicks on opposite corners of a box surrounding the area to zoom.	Ctl-B2
Zoom Cursor (Drag)	Zooms the scene after user drags a box around the area to zoom.	Alt-B2
Pan Cursor	Click on new center of view.	Sh-B3

Changing the View Transformation Button Bindings

The default mapping of functions to mouse buttons, described in Table 3-4 above, can be modified. There are two ways to assign a function to a button/modifier combination.

First, you can use the command

Mouse Function <function_id> Button <1|2|3> [Shift][Control][Alt].

Type **Help Mouse Function** to see a list of function ids that may be used in this command.

Second, you can assign functions interactively. To do so, first put the pointer into a graphics window and then hit the **f** key. On-screen instructions will lead you through the rest of the process.

There is also a simplified function that will map the basic Rotate, Zoom, and Pan functions to unmodified mouse buttons:

Mouse ButtonMap Rotate <rotate_button> Zoom <zoom_button> Pan <pan_button>

This function will not change the bindings to mouse buttons plus modifier keys.

Navigational Drawing Mode

Navigational drawing is entered when a mouse button is pressed in a graphics window, and is exited when the button is released. While performing navigational drawing, the scene can be drawn in a *navigational drawing mode* to speed up rendering; the display is returned to the default drawing mode after the button is released. There are three navigational drawing modes:

- **Wireframe Geometry Mode -** In Wireframe Geometry Mode, any visible mesh disappears and the view changes to a wireframe drawing of the visible geometry. This mode makes transformations faster, and often makes it easier to locate a feature of interest.

  Wireframe Geometry Mode is the default navigational drawing mode, and is enabled at startup. Disable the mode by typing a '**W**' while the mouse is in the graphics window. Reenable the mode by again typing a '**W**' while the mouse is in the graphics window.

  The Wireframe Geometry Mode can also be used temporarily as the default drawing mode by typing a '**P**' while the mouse is in the graphics window. This allows multiple mouse

transformations without having to wait for a complicated scene to be rendered in hiddenline or smoothshade mode after each transformation.

- **Model Bounding Box Mode** - This mode draws a single box representing the bounding box of all existing geometry. This mode is even faster than wireframe geometry mode, but does not indicate the location of individual entities.

  This mode may be active at the same time as any other mode, and overrides the other modes when a mouse button is pressed. This mode is off by default. Switch to this mode by typing a '**B**' while the mouse is in the graphics window. Typing '**B**' again will switch to Body Bounding Box Mode (described below). Typing '**B**' a third time will disable the bounding box modes.

- **Body Bounding Box Mode** - This mode is similar to Model Bounding Box Mode, but draws a separate bounding box for each body in the model. This makes it slightly easier to locate items in the scene, without a substantial penalty in rendering speed. Switch to this mode by typing a '**B**' while in Model Bounding Box mode, and while the mouse is in the graphics window.

It is also possible to remain in the normal drawing mode while navigating with the mouse. To do so, disable wireframe geometry mode and the bounding box modes.

Saving and Restoring Views

After performing view transformations, it may be useful to return to a previous view. A view is restored by setting the graphics camera attributes to a given set of values. The following keys, pressed while the pointer is in the graphics window, provide this capability:

V - Restores the view as it was the last time **Display** was entered.

F1 to F12 - These function keys represent 12 saved views. To save a view, hold down the Control key while pressing the function key. To restore that view later, press the same function key without the Control key.

You can also save a view by entering the command

View Save [Position <1-12>] [Window <window_id>]

The current view parameters will be stored in the specified position. If no position is specified, the view can be restored by pressing **V** in the graphics window. If a position is specified, the view can be restored with the command

View Restore Position <1-12> [Window <window_id>]

These commands are useful in as entries in a .cubit startup file. For example, to always have F1 refer to a front view of the model, the following commands could be entered into a .cubit file:

```
From 0 0 1
At 0 0 0
Up 0 1 0
Graphics Autocenter On
View Save Position 1
```

The first three commands set the orientation of the camera. The fourth command ensures that the model will be centered each time the view is restored. The final command saves the view parameters in position 1. The view can be restored by pressing F1 while the pointer is in a graphics window.

Selecting Entities with the Mouse

Many of the commands in CUBIT require the specification of an entity on which the command operates. These entities are usually specified using an object type and ID (see “Entity Specification” on page 25) or a name. The ID of a particular entity can be found by turning labels on in the graphics and redisplaying; however, this can be cumbersome for complicated models. CUBIT provides the capability to select with the mouse individual geometry, mesh or genesis entities. After being selected, the ID of the entity is reported and the entity is highlighted in the scene. After selecting the entities, other actions can be performed on the selection. The various options for selecting entities in CUBIT are described below, and are summarized in Table 3-5.

Table 3-5: Picking and key press operations on the picked entities.

Key	Action
Ctl-B1	Pick entity of the current picking type.
Sh-Ctl-B1	Add picked entity of the current picking type to current picked entity list.
Tab	Query-pick; pick entity of current picking type which is below the last-picked entity.
N	Lists what entities are currently selected
L	Lists basic information about each selected entity. This is similar to entering a List command for each selected entity
G	Lists geometric information about the selection. As if the List Geometry command were issued for each selected entity. If there are multiple entities selected, a geometric summary of all selected entities is printed at the end, including information such as the total bounding box of the selection
I	Makes the current selection invisible. This only affects entities that can be made invisible from the command line (i.e., geometric entities).
Shift-Z	Zoom in on the current selection.
E	Echo the ID of the selection to the command line.
A	Add the current selection to the picked group. Only geometry will be added to the group (not mesh entities). If a selected entity is already in the picked group, it will not be added a second time.
R	Remove the current selection from the picked group. If a selected entity was not found in the picked group, this command will have no effect.
C	Clear the picked group. The picked group will be empty after this command.
M	Lists what entities are currently in the picked group.
D	Display and select the entities in the picked group.

Entity Selection

Selecting entities typically involves two steps:

1) Specifying the type of entity to select

Clicking on the scene can be interpreted in more than one way. For example, clicking on a curve could be intended to select the curve, a mesh edge owned by the curve, or a sideset containing the curve. The type of entity the user intends to select is called the *picking type*.

In order for CUBIT to correctly interpret mouse clicks, the picking type must be indicated. This can be done in one of two ways. The easiest way to change the picking type is to place the pointer in the graphics window and enter the dimension of the desired picking type and an optional modifier key. The dimension corresponds to the dimension of the objects being picked (0-vertex/node, 1-curve/edge, 2-surface/face, 3-volume/element, 4-body). If a Shift modifier key is held while typing the dimension, the picking type is set to the mesh entity of corresponding dimension, otherwise the geometry entity of that dimension is set as the picking type. For example, typing 2 while the pointer is in the graphics window sets the picking type so that geometric surfaces are picked; typing Shift-1 sets the picking type so that mesh nodes are picked.

The picking type can also be set using the command

Pick <entity_type>

where entity_type is one of the following: **Body**, **Volume**, **Surface**, **Curve**, **Vertex**, **Hex**, **Tet**, **Face**, **Tri**, **Edge**, **Node**, or **DicerSheet**.

2) Selecting the entities.

To select an object, hold down the control key and click on the entity (this command can be mapped to a different button and modifiers, as described in the section on interactive view navigation). Clicking on an entity in this manner will first de-select any previously selected entities, and will then select the entity of the correct type closest to the point clicked. The new selection will be highlighted and its name will be printed in the command window.

Query-Selection

If the highlighted entity is not the object you intended to selected, press the Tab key to move to the next closest entity. You can continue to press tab to loop through all possible selections that are reasonably close to the point where you clicked. Shift-Tab will loop backwards through the same entities.

Multiple Selected Entities

To select an additional entity, without first clearing the current selection, hold down the shift and control keys while clicking on an object. You can select as many objects as you would like. By changing the picking type between selections, more than one type of entity may be selected at a time. When picking multiple entities, each pick action acts as a toggle; if the entity is already picked, it is “unpicked”, or taken out of the picked entities list.

Information About the Selection

When an entity is selected, its name, entity type, and ID are printed in the command window. There are several other actions which can then be performed on the picked entity list. These actions are initiated by pressing a key while the pointer is in the graphics window. Table 3-5

summarizes the actions which operate on the selected entities.

Picked Group

There is a special group whose contents can be altered using picking. This group is named **picked**, and is automatically created by CUBIT. Other than its relationship to interactive

picking, it is identical to other groups and can be operated on from the command line. Like other groups, only geometric entities can be held in the picked group. Table 3-5 lists the graphics window key presses used with the **picked** group.

Note: It is important to distinguish between the current selection and the picked group contents. Clicking on a new entity will select that entity, but will not add it to the picked group. De-selecting an entity will not remove an entity from the picked group.

Substituting the Selection into Commands

There are three ways to use mouse-based selection to specify entities in commands.

• The selection Keyword

You may refer to all currently selected entities by using the word **selection** in a command; the picked type and ID numbers of all selected entities will be substituted directly for **selection**. For example, if Volume 1 and Curve 5 are currently selected, typing

Color selection Blue

is identical to typing

Color Volume 1 Curve 5 Blue.

Note that the **selection** keyword is case sensitive, and must be entered as all lowercase letters.

• Echoing the ID of the Selection

Typing an **e** into a graphics window will cause the ID of each selected entity to be added to the command line at the current insertion point. This is a convenient way to use entities of which you don't already know the name or ID.

As an added convenience, the picking type can be set based on the last word on the command line using the **~** key. For example, a convenient way to set the meshing scheme of a cylinder to sweep would be as follows:

Volume (hit ~, select cylinder, hit e) **Scheme Sweep Source Surface** (hit ~, select endcap, hit e) **Target** (select other endcap, hit e)

The result will be something similar to

Volume 1 Scheme Sweep Source Surface 1 Target 2

Notice that you must use the word Surface in the command, or ~ will not select the correct picking type.

• Using the Picked Group in Commands

Like other groups, the picked group may be used in commands by referring to it by name. The name of the picked group is **picked**. For example, if the contents of the picked group are Volume 1 and Volume 2, the command

Draw picked

is identical to

Draw Volume 1 Volume 2

Note that **picked** is case sensitive, and must be entered as all lowercase letters.

Mesh Slicing

A volume mesh can be viewed one layer at a time using a visualization tool known as mesh slicing. This tool divides the elements of one or more volumes into axis-aligned layers, and then allows the mesh to be displayed one layer at a time. Mesh slicing is especially useful to view the quality of swept meshes that are axis aligned.

Note: Mesh slicing is only intended to be a rough visualization tool. Because the average mesh edge length is used to determine the thickness of each layer, a layer may be more than one element deep. Unstructured meshes, meshes with large variations in edge length, and non-axis-aligned meshes will be more difficult to visualize with this tool.

Mesh slicing can be started either by entering a keypress in the graphics window, which slices the mesh of the entire model, or by entering the command

Graphics Slice {Body | Volume} <id_range> Axis {X | Y | Z}

which slices only the bodies or volumes indicated, with a plane along the axis specified.

Key presses in the graphics window which control mesh slicing are summarized in Table 3-6.

Table 3-6: Mesh slicing key press operations.

Key	Action
X, Y, or Z	Initiate mesh slicing using the X, Y or Z plane, respectively.
J	Move the slicing plane in the positive coordinate direction.
K	Move the slicing plane in the negative coordiante direction.
S	Toggles drawing single or multiple slice layers in the view.
Q	Exit from mesh slicing mode.

Entity Labels

Most entities may be labeled with text that is drawn at the centroid of the entity.

Mesh entities can be labeled with their ID number or their Genesis ID. The command to control labels for mesh entities is

Label {Hex | Face | Edge | Node} <On | Off | Genesis>

Genesis ID labels are only valid after exporting a mesh.

Geometric entities can be labeled with their ID number or with other information using the command

Label {geom_entity_type} <On | Off | Ids | Name [Only | Ids] | Interval | Firmness | Merge | Size | Scheme>

The meaning of each of each label type is listed below. Note that some label types don't make sense for every entity type.

- **On** - The same as Ids.
- **Ids** - The CUBIT ID of the entity.

- **Name** - Name of the entity, if the entity has been named. Default name otherwise.
- **Name Only** - If the entity has been named, use the name as the label. Otherwise, don't use a label.
- **Name ID** - If the entity has been named, use the name as the label. Otherwise, use the ID as the label.
- **Interval** - The number of intervals set on the entity.
- **Firmness** - Same as interval, but followed by a letter indicating the firmness of the interval setting (see chapter 5 for description of firmness settings.)
- **Merge** - Whether or not the entity has been merged.
- **Size** - The mesh size set on this entity.
- **Scheme** - The meshing scheme set for this entity.

Labels for groups of entity types can be turned on or off with the command

Label <All | Geometry | Mesh> <On | Off>

Colors

Color Definitions

CUBIT has a palette of 85 pre-define colors; users may also define their own colors in addition to those define by Cubit. Each color is defined by a name and by its RGB components, which range from 0 to 1.

To define a color, use either of the commands

Color Define "<name>" RGB <r g b>

Color Define "<name>" R <r> G <g> B <b>.

A maximum of 15 user-defined colors may be stored at one time, so it may be necessary to clear a color definition. This is done with the command

Color Release "<color_name>"

Color names can be listed with the command

Help Color

They are also listed in the appendix of this manual, along with their RGB definitions. To view a chart of color names and IDs, including those for user-defined colors, use the command

Draw Colortable

Specifying Colors in Commands

There are three ways to refer to a color in a command. They are

- **ID <id>**
- **name**
- **User "name"**

The first of these three methods may be used for either pre-defined or user-defined colors. The second method is only valid for pre-defined colors, while the third is only valid for user-defined colors. Some examples of specifying colors in commands are:

- By ID - **Color Volume 1 ID 5**

- By Name (Pre-Defined) - **Color Volume 1 Red**
- By Name (User-Defined) - **Color Volume 1 User “mycolor”**

Assigning Colors

Colors can be assigned to all geometric entities except for vertices and to some other objects. To assign a color to an entity or other object, use one of the following commands.

Color <entity_specifier> [Mesh][Geometry] [<color> | default]

Color {NodeSet | SideSet | Block} <id_range> <color>

Color Background <color>.

Color Highlight <color>

Color Axis Text <color>

Color {Curve | Vertex | Hex | Face | Edge | Node | Tet | Tri} Labels <color>

Color Lines <color>

Including the **Mesh** keyword will change the color of the mesh belonging to the specified entity, without changing the color of the entity geometry itself. Conversely, including the **Geometry** keyword will change the geometry color without changing the mesh color. Including both keywords is identical to including neither keyword.

Colors are inherited by child entities. If you explicitly set the color for a volume, for example, all of its surfaces will also be drawn in that color. Once you assign a color to an entity, however, it will remain that color and will no longer follow color changes to parent entities. To make an entity follow the color of its parent after having explicitly set another color, use **Default** as the color name in the color command.

Colors can also be assigned to nodesets, sidesets, and element blocks. These colors do not take effect, however, unless the nodeset, sideset, or element block is drawn with a Draw command.

The background color and the color used to draw highlighted entities can be changed to any color.

By default, the axes are labeled with a white X, Y, and Z, indicating the three primary coordinate directions. If the background is changed to white, these labels are impossible to read; the color used to draw axis labels can be changed to any color. Changing the axis label color will change the text color for both the model axis and the triad (corner axis).

When several entity types are labeled, it can become difficult to determine which labels apply to which entities. To help distinguish which entities are being referred to by the labels, you may want to change the color of labels for specific entity types.

When a meshed surface is drawn in a shaded graphics mode, the mesh edges are not drawn in the same color as the surface. This is to prevent confusion between mesh edges and geometric curves, and to make the mesh edges more visible. The color used to draw mesh edges in this situation is known as the line color, and is gray by default; this color can be changed to any color.

Geometry and Mesh Entity Visibility

The visibility of geometric and mesh entities can be turned on or off, either individually, by entity type, by general entity class (mesh, geometry, etc.), or globally. The commands to set the visibility are:

{geom_list} [Mesh][Geometry] Visibility [On|Off]

{Vertex | Hex | Face | Edge | Node} Visibility [On | Off]

Mesh Visibility [On | Off]

Geometry Visibility [On | Off]

If the **Mesh** keyword is included, only the visibility of the mesh belonging to the specified entity is affected. Similarly, if the **Geometry** keyword is included, only the visibility of the geometry is affected. Including neither keyword is identical to using both keywords.

Invisibility of geometry is inherited; visibility is not. For example, if a volume is invisible, its surfaces are also invisible unless they also belong to some other visible volume. As another case, if the volume is visible, but a surface is set to invisible, the surface will not follow its parent's visibility setting, but will remain invisible.

The visibility of some entity types has special meaning. If hex visibility is on, internal mesh edges become visible. Face visibility only refers to external faces, and faces are only visible when in a shaded graphics mode.

After turning mesh visibility off, all mesh will remain invisible until mesh visibility is turned on again. This is true no matter what other visibility commands are entered.

Similarly, after turning geometry visibility off, all geometry will remain invisible until geometry visibility is turned on again. This is true no matter what other visibility commands are entered.

Graphics Camera

One way to change what is visible in the graphics window is to manipulate the camera used to generate the scene. A scene camera has *attributes* described below, and depicted graphically in Figure 3-4. The values of these camera attributes determine how the scene appears in the graphics window.

- **Position (From)** - The location of the camera in model coordinates.
- **View Direction (At)** - The focal point of the camera in model coordinates.
- **Up Direction (Up)** - The point indicating the direction to which the top of the camera is pointing. The Up point determines how the camera is rotated about its line of sight.
- **Projection** - Determines how the three-dimensional model is mapped to the two-dimensional graphics window.
- **Perspective Angle** - Twice the angle between the line of sight and the edge of the visible portion of the scene.

At any time, the camera can be moved back to its original position and view using the command

View Reset

To see the current settings of these attributes, use the command

List View

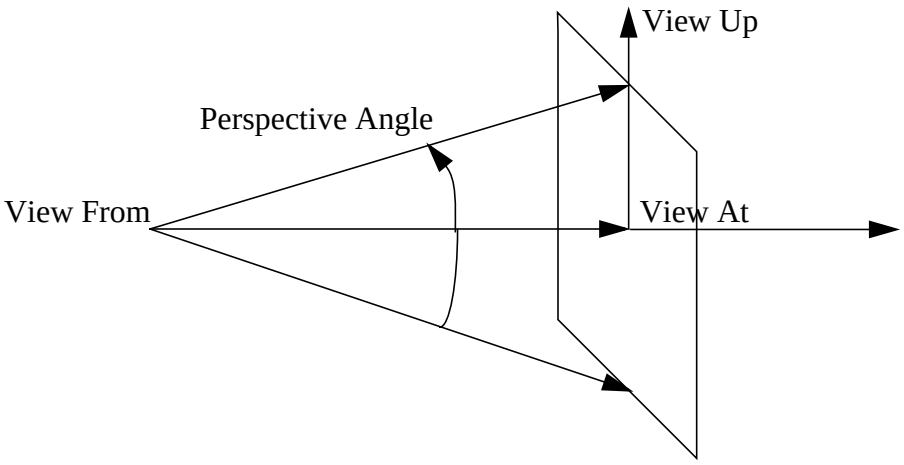


Figure 3-4: Schematic of From, At, Up, and Perspective Angle

The current value of the view attributes will be printed to the terminal window, along with other useful view information such as the current graphics mode and the width of the current scene in model coordinates.

Changing Camera Attributes Using Rotate, Zoom Pan

Commands used to affect camera position or other functions are listed below. All rotation, panning, and zooming operations can include the **Animation Steps** qualifier, makes the image pass smoothly through the total transformation. Animation also allows the user to see how a transformation command arrives at its destination by showing the intermediate positions.

• Rotation

Rotate <degrees> About [Screen | Camera | World] {X | Y | Z} [Animation Steps <number_steps>]

Rotate <degrees> About Curve <curve> [Animation Steps <number_steps>]

Rotate <degrees> About Vertex <vertex_1> Vertex <vertex_2> [Animation Steps <number_steps>]

Rotation of the view can be specified by an angle about an axis in model coordinates, about the camera's "At" point, or about the camera itself. Additionally rotations can be specified about any general axis by specifying start and end points to define the general vector. The *right hand rule* is used in all rotations.

Plain degree rotations are in the **Screen** coordinate system by default, which is centered on the camera's At point. The **Camera** keyword causes the camera to rotate about itself (the camera's From point). The **World** keyword causes the rotation to occur about the model's coordinate system. Rotations can also be performed about the line joining the two end vertices of a curve in the model, or a line connecting two vertices in the model.

• *Panning*

Pan [{Left|Right} <factor1>] [{Up|Down} <factor2>] [Screen | World] [Animation Steps <number_steps>]

Pan Cursor

Panning causes the camera to be moved up, down, left, or right. In terms of camera attributes, the From point and At point are translated equal distances and directions, while the perspective angle and up vector remain unchanged. The scene can also be panned by a factor of the graphics window size.

Screen and **World** indicate which coordinate system **<factor>** is in. If **Screen** is indicated (the default), **<factor>** is in screen coordinates, in which the width of the screen is one unit. If **World** is indicated, **<factor>** is expressed in the model units.

The **Pan Cursor** command is used to indicate the position of the desired view center with the mouse.

• *Zooming*

Zoom Cursor [Click | Drag] [Animation Steps <number_steps>]

Zoom Screen <factor> [Animation Steps <number_steps>]

Zoom <x_min> <y_min> <x_max> <y_max> [Animation Steps <number_steps>]

Zoom {Group | Body | Volume | Surface | Curve | Vertex | Hex | Tet | Face | Tri | Edge | Node} <id_range> [Animation Steps <number_steps>]

Zoom Reset

After entering **Zoom Cursor**, move the cursor to the graphics window. If the **Click** option was entered, click on opposite corners of the desired zoom area; otherwise, drag a box around the area to zoom by holding down the left mouse button until the desired area is boxed in. **Click** is the default option for this command.

Zoom Screen will move the camera **<factor>** times closer to its focal point. The result is that objects on the focal plane will appear **<factor>** times larger.

Zooming on a specific portion of the screen is accomplished by specifying the zoom area in screen coordinates; for example, **Zoom 0 0 .25 .25** will zoom in on the bottom left quarter of the screen.

Zooming on a particular entity in the model is accomplished by specifying the entity type and ID after entering **Zoom**. The image will be adjusted to fit bounding box of the specified entity into the graphics window, and the specified entity will be highlighted.

To center the view on all visible entities, use the **Zoom Reset** command.

Changing Camera Attributes Directly

Camera attributes are most easily modified using interactive mouse manipulation (see “Mouse-Based View Navigation” on page 32) or using the rotate, pan and zoom commands described

above. However, the camera attributes can also be modified directly with the following commands:

From <x y z>

At <x y z>

Up <x y z>

Graphics Perspective <On|Off>

Graphics Perspective Angle <degrees>

If graphics perspective is on, a perspective projection is used; if graphics perspective is off, an orthographic projection is used. With a perspective projection, the scene is drawn as it would look to a real camera. This gives a three-dimensional sense of depth, but causes most parallel lines to be drawn non-parallel to each other. If an orthographic projection is used, no sense of depth is given, but parallel lines are always drawn parallel to each other.

In a perspective view, changing the perspective angle changes the field of view by changing the angle from the line of sight to the edge of the visible scene. The effect is similar to a telephoto zoom with a camera. A smaller perspective angle results in a larger zoom. This command has no effect when graphics perspective is off.

Graphics Windows

Window Size and Position

By default, CUBIT will create a single graphics window when it starts up (to run CUBIT without a graphics window, include **-nographics** on the command line when launching CUBIT.) The graphics window position and size is most easily adjusted using the mouse, like any other window on an X-windows screen. However, the size of the graphics window can also be controlled using the following commands:

Graphics WindowSize <width_in_pixels> <height_in_pixels>

Graphics WindowSize Maximum

In addition, the graphics window size and position can be controlled by placing the following line in the user's .Xdefaults file:

cubit.graphics.geometry XxY xpos ypos

where the **X** and **Y** are window width and height in pixels, respectively, and **xpos** and **ypos** are the offsets from the lower left hand corner.

Using Multiple Windows

You can use up to ten graphics windows simultaneously, each with its own camera and view. Each window has an ID, from 1 to 10, shown in the title bar of the window. Commands that control camera attributes apply to only one window at a time, the *active window.* Currently, the display lists of all windows are identical.

The following commands are used to create, delete, and make active additional graphics windows.

Graphics Window Create [ID]

Graphics Window Delete <ID>

Graphics Window Active <ID>

Hardcopy Output

The easiest way to make an image file of a graphics window is to use a screen capture program outside of CUBIT, such as xv. However, CUBIT also has the ability to save its graphical output to files of various formats. The commands for generating hardcopy output files are:

Hardcopy '<filename>' [Encapsulated | Postscript | Eps] [Color | Monochrome] [Window <window_id=active>]

Hardcopy '<filename>' Pict [xsize <xpixels>] [ysize <ypixels>] [Window <window_id=active>]

Hardcopy '<filename>' Cgm [ansi | cals | cleartext] [Window <window_id=active>]

Hardcopy '<filename>' Hpgl [Landscape | Portrait] [xsize <width> ysize <height>] [window <window_id=active>]

Each of these commands saves the view in the specified window (or the current window), to the specified file, in the format indicated. The file can then be sent to a printer or inserted into another document.

Miscellaneous Graphics Options

In addition to the commands discussed above, there are several other graphics system options in CUBIT that can be controlled by the user.

- **Silhouette Lines**

Some shapes, such as cylinders, are drawn with silhouette lines; these lines don't represent true geometric curves, but help visualize the shape of a surface. Silhouette lines can be turned on or off with the command

Graphics Silhouette [On|Off]

The pattern used to draw silhouette lines can be set using the command

Graphics Silhouette Pattern [solid | dashdot | dashed | dotted | dash_2dot | dash_3dot | long_dash | phantom]

- **LineWidth**

This option controls the width of the lines used in the wireframe and hiddenline displays. The default is 1 pixel wide. The command to set the line width is

Graphics LineWidth <width_in_pixels>

- **Highlight LineWidth**

This option controls the width of the lines used when highlighting an entity. Setting this to a width greater than the global line width often makes it easier to locate highlighted entities. If this setting has not been changed, the line width set in the command above is used. The command to set the highlighting line width is

Highlight LineWidth <width_in_pixels>

- **Text Size**.

This option controls the size of text drawn in the graphics window. The size given in this command is the desired size relative to the default size. The command to set the text size is

Graphics Text Size <size>

- **Point Size**

This option controls the size of points drawn in the graphics window, such as vertices or heads of vectors; alternatively, the size of points representing nodes or vertices can be set independently of the global point size. The commands to set the point sizes are

Graphics Point Size <size>

Graphics [Node | Vertex] Point Size <size>

- **Point Style**

Graphical points are drawn as dots by default. They may be drawn as other symbols, such as plus sign. Point styles for nodes and vertices can also be set independently of the global point style. To change the point style, use the commands

Graphics Point Style <style_integer>

Graphics [Node | Vertex] Point Style <style_integer>

Type **Help Point Style** to see a list of valid style integers and the symbol they represent.

- **Graphics Status**

All graphics commands can be disabled or re-enabled with the command

Graphics [On | Off]

While graphics are off, changes in the model will not appear in the graphics window, and all graphics commands will be ignored. When graphics are again turned on, the scene will be updated to reflect the current state of the model.

- **Model Axis**

The model axis may be drawn in the scene at the model origin. The axis is controlled with the command

Graphics Axis [Type <AXIS | Origin>] [on | off]

The command is used to specify whether the model axis is visible, and to determine how the axis is drawn. If you include **Type Axis**, the axis will be drawn as three orthogonal lines; if you include **Type Origin**, the axis will be drawn as a circle at the model origin.

- **Corner Axis (Triad)**

By default, an axis appears in the corner of the graphics window. This corner axis, also called the triad, can be disabled or re-enabled with the command

Graphics Triad [On | Off]

- **Scene Border**

By default, there is a black border drawn around the scene. To remove or restore the border, use the command

Graphics Border [On | Off]

CHAPTER 3: Environment

▼ Graphics Enhancements

Draw Location On Curve

Some commands require you to specify a location on a curve (i.e., webcutting with a plane normal to a curve). This location can be previewed with the following options:

1) A fraction along the curve from the start of the curve, or optionally, from a specified vertex on the curve.
2) A distance along the curve from the start of the curve, or optionally, from a specified vertex on the curve.
3) An xyz position that is moved to the closest point on the given curve.
4) The position of a vertex that is moved to the closest point on the given curve.

Draw Location On Curve <curve id>
{Fraction <f> | Distance <d> | Position <xval><yval><zval> |
Close_To Vertex <vertex_id>}
[[From] Vertex <vertex_id> (optional for 'Fraction' & 'Distance')]

Draw Plane

The ability to preview a plane prior to webcutting or creating the plane is possible with the following commands:

Draw Plane Vertex <v1_id> [vertex] <v2_id> [vertex] <v3_id>
[[intersecting] Body <id_range>] [extended percentage|absolute <val>]
[color 'color_name']

Draw Plane Surface <surface_id>
[[intersecting] Body <id_range>] [extended percentage|absolute <val>]
[color 'color_name']

Draw Plane {xplane|yplane|zplane} [offset <val>]
[[intersecting] Body <id_range>] [extended percentage|absolute <val>]
[color 'color_name']

Draw Plane Normal To Curve <curve_id>
{fraction <f> | distance <d> | position <xval><yval><zval>
| close_to vertex <vertex_id>}
[[from] vertex <vertex_id> (optional for 'fraction' & 'distance')]
[[intersecting] Body <id_range>] [extended percentage|absolute <val>]
[color 'color_name']

The first passes a plane through 3 vertices, the second uses an existing plane, the third draws a plane normal to one of the global axes, and the fourth draws a plane normal to the tangent of a curve at a location along the curve. By default, the commands draw the plane just large enough to intersect the bounding box of the entire model with minimum surface area. Optionally, you can give a list of bodies to intersect for this calculation. You can also extend the size of the surface by either a percentage distance or an absolute distance of the minimum area size. The default color is blue, but you can specify a different one. See Appendix B of the CUBIT Users Guide for available colors in CUBIT.

Draw Cylinder

The ability to preview a cylinder prior to webcutting is possible with the following command:

Draw Cylinder Radius <val> Axis {x|y|z|Vertex <id_1> Vertex <id_2>|
<xyz values>} [Center <x_val> <y_val> <z_val>]
[[intersecting] Body <id_range>] [extended percentage|absolute <val>]
[color 'color_name']

The cylinder is defined by a radius and the cylinder axis. The axis is specified as a line corresponding to a coordinate axis, the normal to a specified surface, two arbitrary points, or an arbitrary point and the origin. The center point through which the cylinder axis passes can also be specified.

By default, the commands draw the cylinder just large enough to just intersect the bounding box of the entire model. Optionally, you can give a list of bodies to intersect for this calculation. You can also extend the length of the cylinder by either a percentage distance or an absolute distance of the cylinder length. The default color is blue, but you can specify a different one. See Appendix B of the CUBIT Users Guide for available colors in CUBIT.

Entity Parsing

Entity parsing has been extended to allow traversal across geometry and mesh entities. For example, the following commands are now valid:

Draw Node in Surface 3
Draw Surface in Edge 362
Draw Hex in Face in Surface 2
Draw Node in Hex in Face in Surface 2
Draw Edge in Node in Surface 2

▼ Listing Information

The **List** commands print information about the current model and session. There are five general areas: *Model Summary, Geometry, Mesh, Special Entities*, and *CUBIT Environment*. The descriptions of these areas includes example output based on the model generated by the journal file in Table 3-7. The model consists of a 1x2x3 brick meshed with element size 0.1.

List Model Summary

The following commands print identical summaries of the model: the number of entities of each geometric, mesh, and special type; see Table 3-8 for sample output.

List Model

List Totals

List Geometry

The following commands list information about the geometry of the model.

list names [group|body|volume|surface|curve|vertex|all]

list {group | body | volume | surface | curve | vertex} [range] [ids]

list {geom_list} [Geometry|Mesh [Detail]]

list {group | body | volume | surface | curve | vertex} <range> {x|y|z}

The first command lists the names in use, and the entity type and id corresponding to each name. Specifying **all** lists names for all types; other options list names for a specific entity type. The names for an individual entity can be obtained by listing just that entity. Sample output from the **list names surface** command is shown in Table 3-9. This output shows that, for example, Surface 2 has the name '`BackSurface`'.

If **ids** is specified, the second command provides information on the number of entities in the model and their identification numbers. This can be very useful for large models in which several geometry decomposition operations have performed. Sample output from the **list surface ids** command is shown in Table 3-10.

The **range** can be very general using the general entity parsing syntax. Using a **range** and **ids** gives a brief synopsis of the local connectivity of the model, e.g. one can list the ids of the surfaces containing vertex 2; see Table 3-11. An intermediately detailed synopsis can be obtained by placing the range of entities in a group, then listing the group.

The third command provides detailed information for each of the specific entities. This information includes the entity's name and id, its meshing scheme and how that scheme was selected, whether it is meshed and other meshing parameters such as smooth scheme, interval size and count. The entity's connectivity is summarized by a table of the entity's subentities and a list of the entity's supentities. Also, the nodesets, sidesets, blocks, and groups containing the entity are listed.

Specifying **geometry** will additionally list the extent of the entity's geometric bounding box, the geoemtric size of the entity, and, depending on entity type, other information such as surface normal. See also the **list {entities} x** command below.

Specifying **mesh** will additionally list the number of mesh entities of each type interior to the entity and on bounding subentities. **Mesh detail** will list the ids of the mesh entities as well, following the format of the **list ids** command above.

Table 3-12 through Table 3-13 show sample output for each of these options.

The fourth command lists the entities sorted by either the x, y, or z coordinate of their geometric center. For example, in a large, basically cylindrical model centered around z-axis, it is useful to list the surfaces of a volume sorted by z to identify the source and target sweeping surfaces. An example for our toy model appears as Table 3-15, "'List <entities> x' Example.," on page 57.

List Mesh

The following commands list mesh entity information.

list { hex | face | edge | node } <id_range>

list { hex | face | edge | node } <id_range> ids

For both of these commands, the range can be very general, following the general entity parsing syntax. The first command provides detailed information. For an entity, the information includes its id, owning geometry, subentities and supentities. For a hex, the Exodus Id[1] is also listed. For a node, its coordinates are listed. The second command just lists the entity ids, and is usually used in conjunction with complex ranges. Table 3-16 gives examples.

List Special Entities

List {special_type} [range]

Special entities include (element) blocks, sidesets and nodesets (representing boundary conditions), and boundary layers. Like the list geometry and mesh commands, if no range is specified then the number of entities of the given type is summarized. Otherwise, listing a special entity prints the mesh and geometry it contains. Sample output for **list block**, **list sideset**, and **list nodeset** is shown in Table 3-17, Table 3-18, and Table 3-19. (Some special entities are of interest mainly to developers and are not described here, e.g. whisker sheets, whisker hexes, and dicer sheets.)

List CUBIT Environment

The user may list Information about the current CUBIT environment such as *message output settings*, *memory usage*, and *graphics settings*.

Message Output Settings

There are several major categories of CUBIT messages.

- **Info** (Information) messages tell the user about normal events, such as the id of a newly created body, or the completion of a meshing algorithm.
- **Warning** messages signal unusual events that are potential problems.
- **Error** messages signal either user error, such as syntax errors, or the failure of some operation, such as the failure to mesh a surface.
- **Echo** messages tell the user what was journaled.
- **Debug** messages tell developers about algorithm progress. There are many types of Debug messages, each one concentrating on a different aspect of CUBIT.

By default, Info, Warning, Error, and Echo messages are printed, and Debug messages are not printed. Information, Warning and Debug message printing can be turned on or off (or toggled) with a set command; error messages are always printed. Debugging output can be redirected to a file. Current message printing settings can be listed.

List {echo | info | warning | debug }

Set {echo | info | warning } [on|off]

[Set] Debug <index> [on|off]

[Set] Debug <index> File <'filename'>

[Set] Debug <index> Terminal

1. The Exodus Id is the hex's id in an exported Exodus database, not in the CUBIT model. Before writing the Exodus database the Exodus Id appears as -1.

Message flags can also be set using command line options, e.g. **-warning={on|off}** and **-information={on|off}**. Debug flags can be set on with **-debug=<setting>**, where **<setting>** is a comma-separated list of integers or ranges of integers denoting which flags to turn on. E.g. to set debug flags 1, 3, and 8 to 10 on, the syntax is **-debug=1,3,8-10**.

In addition to the major categories, there are some special purpose output settings.

[Set] Logging [on|off] [file <'filename'>]

List Logging

If **logging** is enabled, all echo, info, warning, and error messages will be output both to the terminal and to the logging file.

List Settings

The **List Settings** command lists the value of all the message flags, journal file and echo settings, as well as additional information; see Table 3-20. The first section lists a short description of each debug flag and its current setting. Next come the other message settings, followed by some flags affecting algorithm behavior.

Graphical Display Information

List view prints the current graphics view and mode parameters; See "Graphics" on page 29.

Memory Usage Information

Users are encouraged to use Unix commands such as 'top' to check total CUBIT memory use. Developers may check internal memory usage with the following command:

List Memory ['<object type>']

Without an object type, the command prints memory use for all types of objects.

Table 3-7: Journal file for List Examples

```
brick x 1 y 2 z 3
body 1 size 0.1
mesh volume 1
block 1 volume 1
nodeset 1 surface 1
sideset 1 surface 2
group "my_surfaces" add surface 1 to 3

surface 2 name "BackSurface"
surface 3 name "BottomSurface"
surface 1 name "FrontSurface"
surface 4 name "LeftSurface"
surface 5 name "RightSurface"
surface 6 name "TopSurface"
```

Table 3-8: 'List Model' or 'List Totals' Example

```
CUBIT> list model

Model Entity Totals:
        Geometric Entities:
               2 groups
               1 bodies
               1 volumes
               6 surfaces
              12 curves
               8 vertices

        Mesh Entities:
            6000 hexes
               0 pyramids
               0 tets
            7876 faces
               0 tris
            9854 edges
            7161 nodes

        Special Entities:
               1 element blocks
               1 sideSets
               1 nodesets
```

Table 3-9: 'List Names' Example

```
CUBIT> list names surface
______Name______  Type________Id
     BackSurface  Surface      2
   BottomSurface  Surface      3
    FrontSurface  Surface      1
     LeftSurface  Surface      4
    RightSurface  Surface      5
      TopSurface  Surface      6
```

Table 3-10: 'List Surface [range] Ids' Examples

```
CUBIT> list surface ids
  The 6 surface ids are 1 to 6.
CUBIT> list surf ids
  The 108 surface ids are 192 to 266, 268 to 271, 273 to 301.
```

Table 3-11: Using ‘List’ for Querying Connectivity.

```
CUBIT> list surface in vertex 2 ids
  The 3 entity ids are 1, 5, 6.

CUBIT> group “v2_surfs” equals surface in vertex 2
CUBIT> list v2_surfs
Group Entity ’v2_surfs’ (Id = 3)
It owns 3 entities: 3 surfaces.

        Owned Entities:                      Mesh Scheme   Interval:
      ______Name______  Type________Id    +is meshed   Count     Size
          FrontSurface  Surface      1          map+     1 H      0.1
            TopSurface  Surface      6          map+     1 H      0.1
          RightSurface  Surface      5          map+     1 H      0.1
```

Table 3-12: ‘List Group Mesh Detail’ Example

```
CUBIT> list my_surfaces mesh detail
Group Entity ’my_surfaces’ (Id = 2)
It owns 3 entities: 3 surfaces.

        Owned Entities:                      Mesh Scheme   Interval:      Edge
        ______Name______  Type________Id     +is meshed   Count     Size Length
            FrontSurface  Surface      1           map+     1 H      0.1
             BackSurface  Surface      2           map+     1 H      0.1
          BottomSurface   Surface      3           map+     1 H      0.1

_________________Mesh_Information___________
Element_Type  Interior  Boundary     Total
        Face       700         0       700
        Edge      1300       180      1480
        Node       603       178       781

Note for groups
’interior’ elements are inside one of the entities explicitly in the group and
’boundary’ elements are on a subentity that is not explicitly in the group.

Mesh id ranges follow.
  The 700 interior face ids are 1 to 700.
  The 1300 interior edge ids are 61 to 430, 491 to 860, 921 to 1480.
  The 180 boundary edge ids are 1 to 60, 431 to 490, 861 to 920.
  The 1480 total edge ids are 1 to 1480.
  The 603 interior node ids are 61 to 231, 292 to 462, 521 to 781.
  The 178 boundary node ids are 1 to 60, 232 to 291, 463 to 520.
  The 781 total node ids are 1 to 781.
```

Table 3-13: ‘List Surface Geometry’ Example

```
CUBIT> list surface 1 geometry
Surface Entity 'FrontSurface' (Id = 1)
        Meshed:         Yes
        Mesh Scheme:    map     (default)
        Smooth Scheme:  winslow fixed

        Non-periodic
        Interval Count: 1
        Interval Size:  0.100000
        An odd loop:    No      (Surface 1)
        Block Id:       0

        Total number of curves:  4

                      Mesh Scheme  Arc      Interval:      Edge   Vertex Type
______Name_____    Id  +is meshed  Length   Count    Size  Length Start_T  End_T
        Curve 1     1      equal+       2    20 H     0.1     0.1     1      2
        Curve 2     2      equal+       1    10 H     0.1     0.1     2      3
        Curve 3     3      equal+       2    20 H     0.1     0.1     3      4
        Curve 4     4      equal+       1    10 H     0.1     0.1     4      1

        In Volume 1.
        Contained in NodeSet 1, Group 2
        Bounding Box: x = -0.5 to 0.5 (range 1)
                      y = -1 to 1 (range 2)
                      z = 1.5 to 1.5 (range 0)

        Merge Setting = On

        Surface Area: 2.000000

        Surface Normal: xyz = 0.000000 0.000000 1.000000
        At centroid:    xyz = 0.000000 0.000000 1.500000
```

Table 3-14: ‘List Curve’ Example

```
CUBIT> list curve 1 to 8 by 2
                      Mesh Scheme  Arc      Interval:      Edge   Vertex Type
______Name_____    Id  +is meshed  Length   Count    Size  Length Start_T  End_T
        Curve 1     1      equal+       2    20 H     0.1     0.1     1      2
        In FrontSurface, TopSurface.

        Curve 3     3      equal+       2    20 H     0.1     0.1     3      4
        In FrontSurface, LeftSurface.

        Curve 5     5      equal+       2    20 H     0.1     0.1     5      6
        In BackSurface, TopSurface.

        Curve 7     7      equal+       2    20 H     0.1     0.1     7      8
        In BackSurface, LeftSurface.
```

Table 3-15: ‘List <entities> x’ Example.

```
CUBIT> list surface in volume 1 x
Entities sorted by geometric center x-coordinate:
        ______Name______  Type________Id x_Coordinate
             LeftSurface  Surface      4         -0.5
            RightSurface  Surface      5            0
             BackSurface  Surface      2            0
            FrontSurface  Surface      1            0
           BottomSurface  Surface      3            0
              TopSurface  Surface      6          0.5
```

Table 3-16: ‘List Hex’ Examples

```
CUBIT> list hex 5701
   Hex 5701  --  Exodus ID = -1
     Owned by Volume 1 (Volume 1)
     Contains Faces:  6084, 6979, 7065, 7066, 11, 1901
     Contains Edges:  8200, 82, 8206, 8204, 3783, 3841, 3842, 11, 9043, 79, 9127, 81
     Contains Nodes:  70, 6872, 6901, 71, 12, 1913, 1942, 13

CUBIT> comment ‘find hexes containing both node 70 and 71’
CUBIT> list hex in node 70 and hex in node 71 ids
  The 2 entity ids are 5101, 5701.
```

Table 3-17: ‘List Block’ Example

```
CUBIT> list block
  The 1 block id is 1.

CUBIT> list block 1
  Block 1 contains 6000 unexported 3D element(s) of type HEX8.
  Owned Entities:
        ______Name______  Type________Id Mesh_Elements
                Volume 1  Volume       1          6000

CUBIT> export genesis ‘f.gen’
...
CUBIT> list block 1
  Block 1 contains 6000 exported 3D element(s) of type HEX8.
...
```

Table 3-18: ‘List SideSet’ Example

```
CUBIT> list sideset 1
  SideSet 1 contains 200 exported element sides.
  Owned Entities:
        ______Name______  Type________Id Mesh_Elements
             BackSurface  Surface      2           200
```

Table 3-19: ‘List NodeSet’ Example

```
CUBIT> list nodeset 1
  NodeSet 1: contains 231 nodes.
  Owned Entities:
        ______Name______  Type________Id Mesh_Elements
            FrontSurface  Surface      1           231
```

Table 3-20: Sample Output from ‘List Settings’ Command

```
CUBIT> list settings
Debug Flag Settings (flag number, setting, output to, description):
 1  OFF  terminal           Debug Graphics toggle for some debug options.
 2  OFF  terminal           Whisker weaving information
 3  OFF  terminal           Timing information for 3D Meshing routines.
... (many lines deleted) ...
88  OFF  terminal           General virtual geometry stuff
89  OFF  terminal           Tet meshing warning and debug messages

echo            = On
info            = On
journal         = On, journal file = 'cubit13.jou'
warning         = On
logging         = Off
recording       = Off
keep invalid mesh = Off
default names   = Off
name replacement character = '_', suffix character = '@'
default blocks = Volumes
Matching Intervals is fast, TRUE; multiple curves will be fixed per interation.
Note in rare cases 'slow', FALSE, may produce better meshes.
Match Intervals rounding is FALSE; intervals will be rounded towards the user-
specified intervals.
```

▼ Obtaining Help

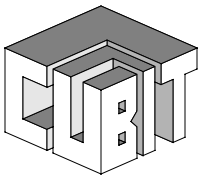
CUBIT can give help on command syntax in two ways. For help on a particular command or keyword, the user can simply type **help <keyword>**. Or, if the user has typed part of a command and is uncertain of the syntax of the remainder of the command, they can type a question mark **?** and help will be printed for all the keywords currently entered. The results of this type of command is shown in Table 3-21.

Table 3-21: Help on Volume & Label

```
CUBIT> volume 3 label fish ?

Help for words: volume & label

Label Volume [ on | off | name [only|id] | id | interval | size |
scheme | merge | firmness ]
```



Chapter 4: Geometry

▼ Introduction

The geometry model in CUBIT serves as the basis for mesh generation, as it describes at the resolution of interest to the analysis. The geometric model is also used for the definition of boundary conditions. The ACIS solid modeling engine is used by CUBIT for solid geometry operations; CUBIT also builds a representation on top of the ACIS model, which it uses to define non-manifold topology. The CUBIT representation is also used as a means for defining virtual geometry, which is used for feature removal and other useful meshing-related tasks.

This chapter begins with definitions of geometric model used in CUBIT and the structure of the nonmanifold geometry represented by CUBIT. This is followed by sections describing geometry import, creation, modification and export.

CUBIT Geometry Model Definitions

Before describing the functionality in CUBIT for viewing and modifying solid geometry, it is useful to give a precise definition of terms used to describe geometry in CUBIT. In this manual, the terms topology and geometry are both used to describe parts of the geometric model. The definitions of these terms are:

Topology: the manner in which geometric entities are connected within a solid model; topological entities in CUBIT include vertices, curves, surfaces, volumes and bodies.

Geometry: the definition of where a topological entity lies in space. For example, a curve may be represented by a straight line, a quadratic curve, or a b-spline. Thus, an element of topology (vertex, curve, etc.) can have one of several different geometric representations.

Topology

Within CUBIT, the topological entities consist of *vertices*, *curves*, *surfaces*, *volumes*, and *bodies*. Each topological entity has a corresponding dimension, representing the number of free parameters required to define that piece of topology. Each topological entity is bounded by one or more topological entities of lower dimension. For example, a surface is bounded by one or more curves, each of which is bounded by one or two vertices.

A CUBIT Body is defined as a collection of other pieces of topology, including curves, surfaces and volumes. While a Body is not required for a complete topological model, it is a convenient mechanism for grouping volumes. Bodies are also used as the basis for solid geometry operations in CUBIT.

Non-Manifold Topology

In many applications, the geometry consists of an assembly of individual parts, which together represent a functioning component. These parts often have mating surfaces, and for typical analyses these surfaces should be joined into a single surface. This results in a mesh on that surface which is shared by the volume meshes on either side of the shared surface. This configuration of geometry is loosely referred to as non-manifold topology[1].

Automatic Detail Suppression

Geometry models often have small features which can be difficult to

1. The definition of non-manifold topology used in the field of Topology is much broader than the definition used here, since it allows additional cases such as dangling faces and edges to exist in the model. Although dangling faces and edges are allowed in the CUBIT geometry model, their use is not common. Unless otherwise stated, the use of the term non-manifold in this manual will refer to the definition given in the text rather than the complete definition known in the field of Topology.

resolve in a mesh. In fact, these features are sometimes too small to see, and are revealed only when the user attempts to mesh the geometry. Automatic detail suppression can be used to remove those features from the meshing model (since virtual geometry operations are used to remove the features, they do not get removed from the actual CAD model).

Small details are identified using the command:

Detail [<ref entity list>] [identify [dimension <dim> [only]]]

After identifying small details, these details can be drawn or removed from the model using the commands:

Detail [<ref entity list>] draw [dimension <dim> [only]]
Detail [<ref entity list>] remove [dimension <dim> [only]]

In the commands above, the dimension option is used to identify the maximum dimension of entities examined for small detail identification (<dim> is 3, 2, 1 for volumes, surface, and curves, respectively). If the only identifier is specified, only entities of the specified dimension are examined, otherwise that dimension and all lower dimensions are examined.

In some cases, details are identified which the user would like to retain in the model; likewise, the algorithm used to identify small details sometimes misses small details the user would like removed from the model. To include or exclude geometric entities *from the list of small details to be removed*, the following command is used:

Detail <ref entity list> [include | exclude]

The algorithm used to identify small details relies on the definition of two quantities, referred to as MEASURSE and SMALL_FRACTION. MEASURE is a measure of the characteristic size of an entity. For geometric curves, this is simply the curve's arc length. For surfaces, MEASURE is computed as the minimum of two quantities, the smallest arc length of curves bounding the surface and the hydraulic

diameter of the surface. (footnote: The hydraulic diameter of a surface is computed as 4.0*A/P, where A is the surface area and P is the summed arc lengths of all bounding curves. For circles, the hydraulic diameter is the circle diameter; for squares, it is the length of the bounding curves. Similarly, for volumes, the hydraulic diameter is computed as 6.0*V/A, which evaluates to the diameter and bounding curve length for perfect spheres and cubes, respectively.) This MEASURE is compared to the user-assigned mesh interval size for that entity; if that ratio is below SMALL_FRACTION, the entity is identified as a small detail. By default, SMALL_FRACTION is set to 1/3.

When removing small curves and surfaces, it is often necessary to composite surfaces together. When done poorly, this results in large variations of surface normal in the region local to the small feature. For this reason, surfaces are not composited unless the dot product of their normal vectors in the neighborhood of the shared curve is greater than COMPATIBLE_FRACTION. By default, this quantity is set to 0.866; this corresponds to an angle between the two surfaces of 180 degrees plus or minus 30 degrees.

In addition to identifying geometrically small entities, the automatic detail suppression algorithm also identifies for removal vertices with valence two. These vertices sometimes appear after performing decomposition on ACIS-based models. Vertices bounding a single curve twice (i.e. the vertex bounding a circular curve) are not removed, however.

▼ Geometry Creation

There are three primary ways of creating geometry for meshing in CUBIT. First, CUBIT provides many geometry primitives for creating common shapes (spheres, bricks, etc.) which can then be modified and combined to build complex models. Secondly, geometry can be imported into CUBIT from an ACIS ".sat" file. Finally, geometry can be defined by building it from the "bottom up", creating vertices, then curves from those vertices, etc. The three methods for creating geometry in CUBIT will be descibed in detail in this section, followed by methods for modifying that geometry.

Geometric Primitives

The geometric primitives supported within CUBIT are pre-defined templates of three-dimensional geometric shapes. Users can create specific instances of these shapes by providing values to the parameters associated with the chosen primitive. Primitives available in CUBIT include the brick, cylinder, torus, prism, frustum, pyramid, and sphere. Figure 4-1 shows a

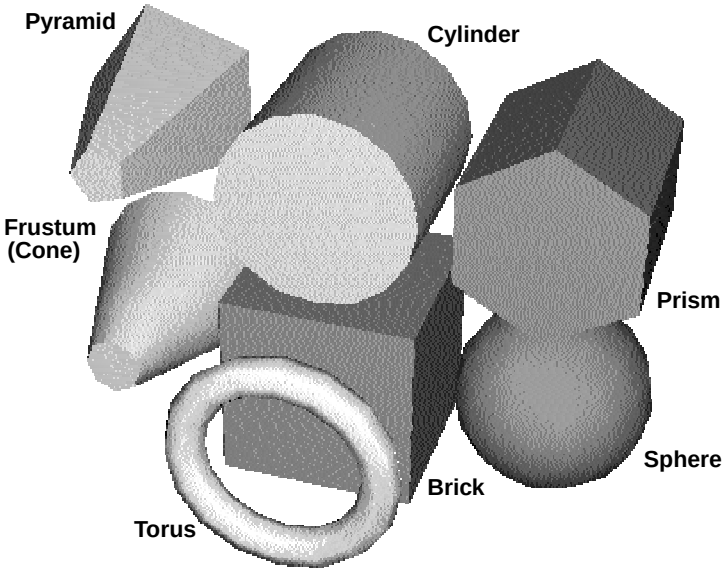


Figure 4-1: Geometry primitives available in CUBIT.

sample of the available primitives. Each primitive, along with the command used to generate it and the parameters associated with it, are described next. For some primitives, several options can be used to generate them, and are described as well.

General Notes

- Primitives are created and given an ID equal to one plus the current highest body ID in the model.
- Primitive solids are created with their centroid at the origin or the world coordinate system.
- For primitives with a Height or Z parameter, the axis going through these primitives will be aligned with the Z axis.
- For primitives with a Major Radius and a Minor Radius, the Major Radius will be along the X axis, the Minor Radius along the Y axis.
- For primitives with a Top Radius, this radius will be that along the X axis; the Y axis radius will be computed using the Major, Minor and Top Radii given.

Brick

The brick is a rectangular parallelepiped.

• Command:

[Create] Brick {Width|X} <width> [{Depth|Y} <depth>] [{Height|Z} <height>]
[Bounding Box entity_type <id_range>]

• Notes:

- A cubical brick is created by specifying only the width or x dimension.
- A brick can be specified to occupy the bounding box of one or more entities, specified on the command line.
- If a bounding box specification is used in conjuction with any of the other paramters (X, Y or Z), the parameters specified override the bounding box results for that or those dimensions.

Cylinder

The cylinder is a constant radius tube with right circular ends.

• Commands:

[Create] Cylinder [height | z] <z-height> radius <x/y-radius>

• Notes:

- A cylinder may also be created using the **frustum** command with all radii set to the same value.

Prism

The prism is an n-sided, constant radius tube with n-sided planar faces on the ends of the tube.

• Commands:

[create] prism [height | z] <z-height> sides <nsides> radius <radius>

[create] prism [height | z] <z-height> sides <nsides> major [radius] <x-radius> minor [radius] <y-radius>

• Notes:

- The radius defines the circumradius of the n-sided polygon on the end caps.
- If a major and minor radius are used, the end caps are bounded by a circum-ellipse instead of a circumcircle.
- The number of sides of a prism must be greater than or equal to three.
- A prism may also be created using the **pyramid** command with all radii set to the same value.

Frustum

A frustum is a general elliptical right frustum, which can also be thought of as a portion of a right elliptical cone.

• Commands:

[create] frustum [height | z] <z-height> major [radius] <x-radius> [minor [radius] <y-radius> top <top-x-radius>]

• Notes:

- If used, Major Radius defines the x-radius and Minor Radius the y-radius.
- If used, Top Radius defines the x-radius at the top of the frustum; the top y radius is calculated based on the ratio of the major and minor radii.

Pyramid

A pyramid is a general n-sided prism.

• Command:

[create] pyramid [height | z] <z-height> sides <nsides> [major [radius] <x-radius> minor [radius] <y-radius>] [top <top-x-radius>]

Sphere

The **sphere** command generates a simple sphere, or, optionally, a portion of a sphere or an annular sphere.

• Commands:

[create] sphere radius <radius> [xpositive] [ypositive] [zpositive] [delete] [inner [radius] <radius>

• Notes:

- If Xpositive, Ypositive, and/or Zpositive are used, a sphere which occupies that side of the coordinate plane only is generated, or, if the delete keyword is used, the sphere will occupy the other side of the coordinate plane(s) specified. These options are used to generate hemisphere, quarter sphere or a sphere octant (eighth sphere).
- If the inner radius is specified, a hollow sphere will be created with a void whose radius is the specified inner radius.

Torus

The torus command generates a simple torus.

• Command:

[create] torus major [radius] <major-radius> minor [radius] <minor-radius>

• Notes:

- **Minor Radius** is the radius of the cross-section of the torus; **Major Radius** is the radius of the spine of the torus.
- The minor radius must be less than the major radius.

Importing Geometry

CUBIT can import geometry in the ACIS “sat” file format. For compatability with Sandia legacy applications, FASTQ input decks can also be used to create geometry.

There are many ways to get geometry into the ACIS format, depending on where that geometry is created. If the geometry is constructed inside CUBIT, the model can be exported directly to an ACIS file (see). If the model has been constructed in Pro/Engineer, there are several methods for translating Pro/E files to the ACIS format[1]. ACIS files can also be exported directly from several commercial CAD packages, including SolidWorks™, AutoCAD, and HP PE/SolidDesigner.

Importing ACIS Models

1. For information about translating Pro/Engineer files into ACIS format, contact the CUBIT group at cubit-dev@sandia.gov, or see the web page http://sass2248.jal.sandia.gov/... on the Sandia Internal Restricted Network.

The command used to read an ACIS file is:

Import Acis '<acis_sat_filename>'

ACIS files can also be imported using the "-solid" command-line option (see "Executing CUBIT" on page 4 for details.) Note that the filename must be enclosed in single or double quotes. This command will create as many bodies within CUBIT as there are bodies in the input file.

Importing FASTQ Models

Support is available for reading a FASTQ file directly into CUBIT. FASTQ files are imported into CUBIT using the command:

Import Fastq '<fastq_filename>'

Note that the filename must be enclosed in single or double quotes.

All FASTQ commands are fully supported except for the **Body** command (which is ignored, if present, as it is unnecessary), the "corn" (corner) line type, and some of the specialized mapping primitive **Scheme** commands. Standard mapping, paving, and triangle primitive scheme commands are handled. The pentagon, semicircle, and transition primitives are not handled directly, but are meshed using the paving scheme. The FASTQ input file may have to be modified if the **Scheme** commands use any non-alphabetic characters such as '+', '(', or ')'. Circular lines with non-constant radius are generated as a logarithmic decrement spiral in FASTQ; in CUBIT they will be generated as an elliptical curve.

Since a FASTQ file by definition will be defined in a plane, it must be projected or swept to generate three dimensional geometry. CUBIT supports sweeping options to convert imported FASTQ geometries into volumetric regions.

Importing ExodusII Files

ExodusII finite element data files can be imported under certain conditions. The capability to generate new geometry from deformed mesh is available for 2D ExodusII files (4, 8, or 9 node QUAD or SHELL element types) that do not have enclosed voids (holes surrounded by mesh) and which were originally generated with CUBIT and exported to ExodusII with the **Nodeset Associativity** option set to on. The **Nodeset Associativity** command records the topology of the geometry into special nodesets which allow CUBIT to reconstruct a new solid model from the mesh even after it has been deformed. The new solid model of the deformed geometry can be remeshed with standard techniques or meshed with a sizing function that can also be imported into CUBIT from the same ExodusII file. CUBIT's implementation of the paving algorithm can generate a mesh following a sizing function to capture a gradient of any variable (element or nodal) present in the ExodusII file.

For more details on importing mesh for geometry, including command syntax, see ... (chapter 5).

Bottom-Up Geometry Creation

CUBIT supports the ability to create geometry from a collection of lower order entities. This is accomplished by first creating vertices, connecting vertices with curves and connecting curves into surfaces. Currently bodies or volumes may not be constructed by stitching a set of surfaces together, however surfaces may be swept or rotated to create bodies or volumes (see "Volume" on page 68). Existing geometry may be combined with new geometry to create higher order entities. For example, a new surface can be created using a combination of new curves and

curves already extant in the model. Commands and details for creating each type of geometry entity are given below.

Vertex

The commands available for creating new vertices directly in CUBIT are:

Create Vertex <x><y><z> [on [Curve | Surface] <id>]

Create Vertex <fraction> from Vertex <id> on Curve <id>

A vertex can be created which lies on a curve or surface in the geometric model by specifying the curve or surface id; the position of the vertex will be the point on the specified entity which is closest to the position specified on the command.

A vertex can be positioned a certain fraction of the arc length along a curve using the second form of the command.

Curve

Curves are created by specifying the bounding lower-order topology (i.e. the vertices) and the geometry (shape) of the curve (along with any parameters necessary for that geometry). There are three forms of this command:

Create Curve [Vertex] <vertex_id> [Vertex] <vertex_id> [On Surface <surface_id>]

Create Curve [Vertex] <vertex_id> [Vertex] <vertex_id> [Vertex] <vertex_id> [Parabolic]

Create Curve from Curve <curve_id>

The first form of the command creates a straight line or a line lying on the specified surface. If a surface is used, the curve will lie on that surface but will not be associated with the surface's topology.

A parabolic curve is created by specifying a third vertex, which completes the definition of a parabolic arc which goes through the three vertices.

The third form of the command actually copies the geometric definition in the specified curve to the newly created curve. The new curve is free floating.

In all cases, the specified vertices are not used directly but rather their positions are used to create new vertices.

Surface

Surfaces are created in CUBIT by fitting an analytic or spline surface over a set of bounding curves. The curves must form a closed loop and only one loop of curves may be supplied. The result is a "sheet body" or a body that has zero measurable volume (it does however have a volume entity). Booleans and special webcutting commands may be used with to decompose this body or to use it for decomposing other bodies. Booleans can be used to cut holes out of these surfaces.

There are three forms of the create surface command:

Create Surface Curve <curve_id_1> <curve_id_2> <curve_id_3>...

Create Surface from Surface <surface_id>

Create Surface extended from Surface <surface_id>

The first form of this command produces an analytic or spline surface fit to cover the bounding curves.

The second form creates a surface using the same geometric description of the specified surface. The new surface will be a stand-alone sheet body that is geometrically identical to the user supplied surface.

The third form of the command creates a surface that is extended from a given surface. The specified surface's geometry is examined and extended out "infinitely" relative to the current model in CUBIT (i.e. extended to just beyond the bounding box of the entire model). The given surfaces are extended as shown in Table 4-1.

Table 4-1: Surface Extension Results

Given Surface Type	Resulting Extended Surface
Spherical	Shell of Full Sphere
Planar	Plane of infinite size relative to model
Toroidal	Shell of Full Taurus
Conical, cone, cylinder…	Shell of outside conic axially aligned with given conic of infinite height relative to model
Spline	Surface is extended to extents of the spline definition. This may not be any further than the surface itself, so caution should be used here.

Volume

Currently, CUBIT can create volumes from surfaces only by sweeping a single surface into a 3D solid. Sweeping of planar surfaces, belonging either to two- or three-dimensional bodies, is allowed - non-planar faces are not supported at this time.

There are two forms of the sweep command; the syntax and details for each are given below. In both forms, the optional **draft_angle** parameter specifies the angle at which the lateral faces of the swept solid will be inclined to the sweep direction. It can also be described as the angle at which the profile expands or contracts as it is swept. The default value is 0.0. The optional **draft_type** parameter is an ACIS-related parameter and specifies what should be done to the corners of the swept solid when a non-zero draft angle is specified. A value of 0 is the default value and implies an extended treatment of the corners. A value of 1 is also valid and implies a rounded (blended) treatment of the corners.

sweep surface {<surface_id_range> | all} vector <x_vector y_vector z_vector> [distance <distance_value>] [draft_angle <degrees>] [draft_type <0 | 1>]

Sweeps a surface a specified distance along a specified vector. Specifying the distance of the sweep is optional; if this parameter is not provided, the face is swept a distance equal to the length of the specified vector.

sweep surface {<surface_id_range> | all} axis {<xpoint ypoint zpoint xvector yvector zvector> | xaxis | yaxis | zaxis} angle <degrees> [steps <number_of_sweep_steps>] [draft_angle <degrees>] [draft_type <0 | 1>]

Sweeps a surface about a specified vector or axis through a specified angle. The **axis** of revolution is specified using either a starting point and a vector, or by a coordinate axis. This axis must lie in the plane of the surfaces being swept. The **steps** parameter defaults to a value of 0 which creates a circular sweep path. If a positive, non-zero value (say, *n*) is specified, then the sweep path consists of a series of *n* linear segments, each subtending an angle of *[(sweep_angle) / (steps-1)]* at the axis of revolution.

Note: Specifying multiple surfaces that belong to the same body will not work as expected, as ACIS performs the sweep operation *in place*. Hence, if a range of surfaces is provided, they ought to each belong to different bodies.

The sweep operations have been designed to produce valid solids of positive volume, even though the underlying solid modeling kernel library that actually executes the operation, ACIS, allows the generation of solids of negative volume (i.e., voids) using a sweep.

Geometry Transforms

Bodies can be modified in CUBIT using transform operations, which include align, copy, move, reflect, restore, rotate, and scale. With the exception of the copy operation, transform operations in CUBIT do*not* create new topology, rather they modify the geometry of the specified bodies.

Align

The align command is a combination of the rotate and move commands. The align command will align the surface of a given body with any other surface in the model, such that the surface centroids are coincident and the normals are pointing either in the same or opposite direction (depending on their initial alignment.) The syntax of this command is:

Align Body <body_id> Surface <surface_id> with Surface <body_id>

This transformation is useful for aligning surfaces in preparation for geometry decomposition.

Copy

The copy command copies an existing body to a new body without modifying the existing body. A copy can be made of several bodies at once, and the resulting new bodies can be translated or rotated at the same time. The commands for copying bodies are:

Body <range> copy [move <x-offset> <y-offset> <z-offset>]

Body <range> copy [reflect {x | y | z}]

Body <range> copy [reflect <x-comp> <y-comp> <z-comp>]

Body <range> copy [rotate <angle> about {x | y | z}]

Body <range> copy [rotate <angle> about <x-comp> <y-comp> <z-comp>]

Body <range> copy [scale <scale-factor>]

If the **copy** command is used to generate new bodies, a copy of the original mesh generated in the original body can also be copied directly into the new body. This is currently limited to copies that do not interact with adjacent geometry through non-manifold topology. For details on mesh copies, see “Mesh Importing and Duplicating” on page 167.

Move

The **move** command moves a body by a specified offset. The commands to move bodies are:

Body <id_range> [Copy] Move <dx> <dy> <dz>

Body <id_range> [Copy] Move {x|y|z} <distance>

Move {Body|Group} <id_range> Normal To Surface <id> Distance <val>

Move {Body|Group} <id_range> XYZ <x_val> <y_val> <z_val>

If the **copy** option is specified, a copy is made and the copy is moved by the specified offset.

It is also possible to move bodies to locations specified either absolutely or relative to other geometry entities in the model:

Move {entity} <id_range> location entity <id> [except [x] [y] [z]]

Move {entity} <id_range> location [x <val>] [y <val>] [z <val>] [except [x] [y] [z]]

Here entity is {vertex|curve|surface|volume|body}, and any combination of entities may be specified (in this case, the body containing the specified entity is moved). This command moves the center of the entities to the specified location. (Note that bodies are integral, so moving an entity also moves all other entities that are in the same body.) “Except” is used to preserve the x, y, or z plane in which the center of the entity lies.

Scale

The **scale** command resizes the body by a constant scale. The body will be scaled about its centroid. The command to scale bodies is:

body <range> [copy] scale <scale>

If the **copy** option is specified, a copy is made and scaled the specified amount.

Rotate

The **rotate** command rotates a body about a given axis without adding any new geometry. If the **Angle** or any **Components** are not specified they are defaulted to be zero. The commands to rotate a body or bodies are:

body <range> [copy] rotate <angle> about {x | y | z}

body <range> [copy] rotate <angle> about <x-comp> <y-comp> <z-comp>

Rotate {Body|Group} <id_range> Angle <val> Axis {X|Y|Z|Normal of Surface <id>| Vertex <id_1> Vertex <id_2>}

If the **copy** option is specified, a copy is made and rotated the specified amount.

Reflect

The **reflect** command mirrors the body about a plane normal to the vector supplied. The reflect command will *destroy* the existing body and replace it with the new reflected body, unless the copy option is used.

body <range> [copy] reflect <x-comp> <y-comp> <z-comp>

body <range> [copy] reflect {x | y | z}

Restore

The restore command removes all previous geometry transformations from the specified body. The command to restore bodies is:

body <range> restore

▼ Geometry Booleans

Boolean operations are ones that modify the geometry and/or the topology of existing solids. Boolean operators supported in CUBIT include imprint, intersect, separate, section, subtract, and union. These operations usually replace the original bodies input to the boolean with new ones.

Imprint

To produce a non-manifold geometry model from a manifold geometry, coincident surfaces must be merged together (see “Geometry Merging” on page 80); this merge can only take place if the surfaces to be merged have like topology and geometry. While various parts of an assembly will typically have surfaces which coincide geometrically, an imprint is necessary to make the surfaces have like topology.

The commands used to imprint bodies together are:

Imprint <body1_id> with <body2_id>

Imprint [Body] All

An Imprint All will imprint all bodies in the model pairwise; bounding boxes are used to filter out imprint calls for bodies which clearly don’t intersect.

Intersect

The **intersect** command generates a new body composed of the space that is shared by the two bodies being intersected. Both of the original bodies will be deleted and the new body will be given the next highest body ID available. The command is:

Intersect <body1_id> with <body2_id> [keep]

The **keep** option results in the original bodies used in the intersect being kept.

Section

This command will cut a body or group of bodies with a plane, keeping geometry on one side of the plane and discarding the rest. The syntax for this command is:

Section {body|group} <id_range> {xplane|yplane|zplane} [offset <value>] [reverse] [keep]

Section {body|group} <id_range> surface <id> [reverse] [keep]

In the first form, the specified coordinate plane is used to cut the specified bodies. The offset option is used to specify an offset from the coordinate plane. In the second form, an existing (planar) surface is used to section the model. In either case, the reverse keyword results in discarding the positive side of the specified plane or surface instead of the other side. The **keep** option results in keeping both sides; the section command used with this option is equivalent to webcutting with a plane.

Separate

The separate command is used to separate a body with multiple volumes into a multiple bodies with single volumes. The command is:

Separate Body {id_range|all}

Subtract

The subtract operation subtracts one body from another set of bodies. The order of subtraction is significant - the body or bodies specified before the From keyword is/are subtracted from bodies specified after From. Both of the original bodies are deleted and the new body is given the next highest body ID available, unless the **keep** keyword is given. The command is:

Subtract <body1_id> from [Body] <body_id_range> [keep]

Unite

The unite operation combines two or more bodies into a single body. The original bodies are deleted and the new body is given the next highest body ID available, unless the **keep** option is used. The commands are:

Unite <body1_id> with <body2_id> [keep]

Unite Body {<range> | all} [keep]

The second form of the command unites multiple bodies in a single operation. If the **all** option is used, all bodies in the model are united into a single body. If the bodies that are united do not overlap or touch, the two bodies are combined into a single body with multiple volumes.

Geometry Decomposition

Geometry decomposition is often required to generate an all-hexahedral mesh for three-dimensional solids, as fully automatic all-hex mesh generation of arbitrary solids is not yet possible in CUBIT. While geometry booleans can be used for decomposition (and are the basis of the underlying implementation of advanced decomposition tools described here), CUBIT has a webcut capability specially tuned for decomposition.

Web Cutting

The term “web cutting” refers to the act of cutting an existing body or bodies, referred to as the “blank”, into two or more pieces through the use of some form of cutting tool, or “tool”. The two primary types of cutting tools available in CUBIT are surfaces (either pre-existing surfaces in the model or infinite or semi-infinite surfaces defined for webcutting), or pre-existing bodies.

The various forms of the webcut command can be classified by the type of tool used for cutting. These forms are described below, starting with the simplest type of tool and progressing to more complex types.

Webcut Using Planar or Cylindrical Surface

The commands used to webcut with a planar surface in CUBIT are:

webcut {blank} plane {xplane | yplane | zplane} [offset <value>]

webcut {blank} plane surface <surface_id>

webcut {blank} plane vertex <id> vertex <id> vertex <id>

webcut {blank} cylinder radius <val.> axis {x|y|z|normal of surface <id>| vertex <id_1> vertex <id_2>| <x_val> <y_val> <z_val>>} [center <x_val> <y_val> <z_val>]

[NOIMPRINT|imprint][NOMERGE|merge] [group_results]

In the command’s simplest form, a coordinate plane can be used to cut the model, and can optionally be offset a positive or negative distance from its position at the origin.

An existing planar surface can also be used to cut the model; in this case, the surface is identified by its ID as the cutting tool.

Finally, any arbitrary planar surface can be used by specifying three vertices which define the plane.

A semi-infinite cylindrical surface can be used by specifying the cylinder radius, andthe cylinder axis. The axis is specified as a line corresponding to a coordinate axis, the normal to a specified surface, two arbitrary points, or an arbitrary point and the origin. The “center” point through which the cylinder axis passes can also be specified.

Webcut with Arbitrary Surface

An arbitrary “sheet” surface can also be used to webcut a body. This sheet need not be planar, and can be bounded or infinite. The following commands are used:

webcut {blank} with sheet {body|surface} <id> [webcut_options]

webcut {blank} with sheet extended from surface <id> [webcut_options]

In its first form, the command uses a sheet body, either one that is pre-existing or one formed from a specified surface. Note that in this latter case the (bounded) surface should completely cut the body into two pieces. Sheet bodies can be formed from a single surface, but can also be the combination of many surfaces; this form of webcut can be used with quite complicated cutting surfaces.

Extended sheet surfaces can also be used; in this case, the specified surface will be extended in all directions possible. Note that some spline surfaces are limited in extent, and so these surfaces may or may not completely cut the blank.

Webcut Using Tool Body

Any existing body in the geometric model can be used to cut other bodies; the command to do this is:

webcut {blank} tool [body] <id> [webcut_options]

This simply uses the specified tool body in a set of boolean operations to split the blank into two or more pieces.

Webcut Options

The following options can be used with all webcut commands:

Group_results: The various pieces resulting from the previous command are placed into a group named ‘webcut_group’.

[Imprint | Noimprint]: In its default implementation, webcutting results in the pieces not being imprinted on one another; this option forces the code to imprint the pieces after webcutting.

[Merge | Nomerge]: By default, the pieces resulting from an imprint are manifold; specifying this option results in a merge check for all surfaces in the pieces resulting from the webcut.

General Notes

The primary purpose of web cutting is to make an existing model meshable with the hex meshing algorithms available in CUBIT. While web cutting can also be used to build the initial geometric model, the implementation and command interface to web cutting have been designed to serve its primary purpose. Several important things to remember about webcutting are as follows:

- The geometric model should be checked for integrity (using imprinting and merging) before starting the decomposition process. This makes the checking process easier, since there are fewer bodies and surfaces to check. Once the model passes that initial integrity check, it is rare that decompositions using webcut will result in a model that does not also pass the same checks.
- The use of the Imprint option can in cases save execution time, since it limits the scope of the imprint operations and thereby works faster. The alternative is performing and Imprint All on the pieces of the model after all decompositions have been completed; this operation has been made much faster in more current releases of CUBIT, but will still take a noticeable amount of time for complicated models.
- While the Webcut commands make it very simple to cut your model into very many pieces, we recommend that the user restrict the decomposition they perform to only that necessary for meshability or for obtaining an acceptable mesh. Having more volumes in the model may

simplify individual volumes, but may not always result in a higher quality mesh; it will always increase the run time and complexity of the meshing task.

Appendix B contains some examples that demonstrate the use of webcutting operations.

Split Periodic

Solids which contain periodic surfaces include cylinders, torii and spheres. Splitting periodic surfaces can in some cases simplify meshing, and will result in curves and surfaces being added to the volume. The command used to split periodic surfaces is:

Split Periodic Body {id_range|all}

This command splits all periodic surfaces in a body or bodies.

▼ Virtual Geometry:

Modify topology of the model within Cubit without affecting geometry and without making changes to the actual solid model. All Virtual Geometry (VG) operations are reversible (original solid model topology can be restored.)

General Notes:

Operations on the solid model cannot be done for bodies containing virtual geometry. Solid modeling operations (webcutting, imprinting, booleans, etc.) must be done prior to creating virtual geometry.

Virtual geometry operations cannot be done on meshed geometry.

Compositing:

Combine a set of connected curves into a single composite curve, or a set of connected surfaces into a single surface. The general purpose is to suppress (remove) the child geometry common to those entities being composited. For example, compositing a set of curves suppresses the vertices common to those curves, thus removing the constraint that a node must be placed at that vertex location.

The basic form of the command to create composites is:

composite create {surface|curve} <id_list>

This command will composite as many surfaces (or curves) as possible, possibly creating multiple composites. The command to remove a composite is:

composite delete {surface|curve} <id>

Compositing over large C1 discontinuities may confuse meshing algorithms and/or result in a bad mesh.

Composite Curves:

The full command for the creation of composite curves is:

Composite Create Curve <id_range> [keep vertex <id_list>] [angle <degrees>]

The additional arguments provide two methods to prevent vertices from being removed from the model (composited over.) The first method, "keep vertex" explicitly specifies vertices which are not to be removed. This option can also be used to control which vertex is kept when compositing a set of curves results in a closed curve. The 'angle' option specifies vertices to

keep by the angle between the tangents of the curves at that vertex. A value less than zero will result in no composite curves being created. A value of 180 or greater will result in all possible composites being created. The default behavior is an empty list of vertices to keep, and an angle of 180 degrees.

Composite Surfaces:

The general command for composite surface creation is:

Composite Create Surface <id_range> [angle <degrees>] [nocurves] [keep [angle <degrees>] [vertex <id_list>]]

The first angle argument (the only one if the 'keep' keyword is not present, or the one preceding the 'keep' keyword) prevents curves from being removed from the model (composited over) by specifying the maximum angle between the normals of surfaces adjacent to the curve.

When a composite surface is created, the default behavior is to also composite curves on the boundary of the new composite surface. Curves are automatically composited if the angle between tangents at the common vertex is less than 15 degrees. The 'nocurves' option can be used to prevent any composite curves from being created. The 'keep' keyword can be used to change the default choice of which curves to composite. The arguments following the 'keep' keyword behave the same as for explicit composite curve creation. The 'nocurves' and 'keep' arguments are mutually exclusive.

Other Composite Surface Notes:

It typically takes longer to mesh a single composite surface than to mesh the surfaces used in the creation of the composite.

Composite surfaces uses an approximation method to evaluate the closes point to a trimmed surfaces because it is faster. However, for some highly convoluted surfaces (used to create the composite), this method may return bad results. The command

composite closest_pt surface <id> {gme|emulate}

can be used to disable this behavior. The default is to use the 'emulate' method, as it is typically much faster. Specifying the 'gme' option will force the specified composite surface to use the exact calculation of the closest point to a trimmed surface, as provided by the solid modeler. However, this is considerably slower in most cases.

The "composite create surface" command is non-deterministic in some circumstances. When three or more adjacent surfaces are to be composited, all the surfaces cannot be composited into a single surface, but different subsets of the surfaces may be composited, the command will choose arbitrary subsets to composite. As an example, there are three surfaces A, B, and C, all adjacent to each other. The common curve between A and B is AB, the common curve between B and C is BC, and the common curve between A and C is AC. If the curve BC cannot be removed, either due to the angle specified in the composite command, or because there is a fourth surface, D, also using that curve, the command will arbitrarily choose to either composite A and B or A and C.

Partitioning:

Partitioning provides a method to introduce additional topology into the model, to better constrain meshing algorithms. This is accomplished by splitting, or partitioning, existing curves or surfaces.

Partitioning Curves:

There are three forms to the command to partition a curve:

partition create curve <id> fraction <f1> [<f2> <f3> ...]

partition create curve <id> position <x> <y> <z> [position <x2> <y2> <z2> ...]

partition create curve <id> vertex <id_list>

The first two forms of the command create additional vertices and use those vertices to split a curve. The third form of the command uses existing vertices to split the curve.

Using the 'fraction' option, vertices are created at the specified fractions along the curve (in the range [0,1]). Subsequently, the curve is split at each vertex, resulting in n+1 new curves, where n is the number of fraction values specified.

Using the 'position' option, vertices are created at the closest location along the curve to each of the specified position. Subsequently, the curve is split at each vertex, resulting in n+1 new curves, where n is the number of positions specified.

Curves can also be partitioned via the 'virtual' command discussed below.

Partitioning Surfaces:

There are two forms of the command to partition a surface:

partition create surface <id> curve <id_list>

partition create surface <id> vertex <id1> <id2> [<id3>]

The first form of the command splits the existing surface into several surfaces using the passed list of curves. The end vertices of the curves must be part of the surface. If the vertices are not already part of the surface, they can be made to be part of the surface by partitioning curves in the surface using end vertices of the passed list of curves. Any curves which do not complete loops on the surface are disregarded.

The second form of the command creates a curve (or two curves) and splits the surface using those curves. At most three vertices may be specified. At least two of those vertices must be part of the surface. The third, if specified, must lie on the surface.

Removing partitions:

There are two commands used to remove partitions:

partition merge {curve|surface} <id_list>

partition delete {curve|surface} <id>

The first command combines existing partitions where possible. This command is similar to the 'composite create' command. The difference is that this command is special-cased for partitions, and will result in more efficient geometric evaluations. If all the partitions of a real solid model entity are merged (such that there is only one 'partition' remaining) the virtual geometry will be removed, and the original solid model geometry will be restored to the model.

The second form of the command takes a single partitioned surface or curve, finds all other partitions of the same real geometry, and does the equivalent of the 'partition merge' command for that set of partitions.

Creating New Virtual Geometry (The 'virtual' Command.)

The 'virtual' command provides a method for introducing new geometry for use in defining locations at which to partition. Geometry created with this command exists only in Cubit, not in the actual solid model. Geometry created by the solid modeling engine may also be used to define partition locations.

Virtual Vertices:

Virtual vertices are typically used to define locations at which to partition a curve, and in defining the end points of virtual curves. There are five forms of the command to create a virtual vertex:

virtual create vertex position <x> <y> <z> [position <x1> <y1> <z1>...]

virtual create vertex curve <id> fraction <f> [<f2> <f3> <f4> ...] [nocurves]

virtual create vertex curve <id> position <x> <y> <z> [position <x1> <y1> <z1> ...] [nocurves]

virtual create vertex curve <id> vertex <id_list> [nocurves]

virtual create vertex surface <id> position <x> <y> <z> [position <x1> <y1> <z1> ...]

The first, and simplest form of the command creates vertices at the specified positions. The second, third, and fourth forms of the command create vertices on a curve. They also automatically partition the curve at those vertex locations, as this is generally why the vertices are being created. The 'nocurves' option prevents the automatic curve partitioning. The second and third forms of the command behave the same as the similar curve partitioning commands (unless the 'nocurves' option is present.) The fourth form of the command creates vertices on the specified curve at the closest location on that curve to the specified vertices. The fifth, and final form of the command creates vertices at the closest point(s) to the specified surface from the passed position(s).

Virtual Curves:

Virtual curves are typically used to define locations at which to partition surfaces. There are two forms of the command to create virtual curves:

virtual create curve vertex <id1> <id2>

virtual create curve vertex <id1> <id2> surface <id>

The first form of the command creates a linear curve between the two specified vertices. The second form of the command creates a curve between two vertices, and lying on the specified surface.

Deleting Virtual Vertices and Curves:

Virtual vertices and curves may be deleted using the general Cubit 'delete' command, and are subject to the same restrictions as other geometry. They may not be deleted if they have parent geometry.

Using The 'delete' Command With Composites.

If the general 'delete' command is invoked for a composite surface, the composite surface will be removed, and the original surfaces used to define the composite will be restored to the model. The defining surfaces are NOT also deleted. As with any other surface, the delete command will fail if the composite has a parent volume. This is why the 'composite delete' command is provided. The behavior is analogous for composite curves.

If the delete command is used on a volume containing a composite surface or curve, or on a surface containing a composite curve, the entire volume or surface will be deleted, including the original entities used to define the composite, as those entities are also children of the entity being deleted.

Using the 'delete' Command With Partitions.

It is recommended that the 'delete' command not be used with partitions, as it may break subsequent usage of the 'merge' and 'delete' forms of the 'partition' command for other

partitions of the same real geometry entity. However, if the 'delete' command is used for partitions, the behavior is to delete the specified partition, and when the last partition of the real geometry is deleted, to restore the original geometry.

It works fine to use the delete command on parents of partitions, for example a volume containing partitioned surfaces, or a surface containing partitioned curves. In this case, the Cubit will behave exactly as expected. The specified entity will be deleted along with all of its children, including the partition entities, and the original entities that were partitioned.

▼ Automatic Geometry Decomposition

In many cases, model geometry includes protrusions which, when cut off using geometry decomposition, are easily meshable with existing algorithms. CUBIT includes a feature-based decomposition capability which automates this process. This algorithm operates by finding concave curves in the model, grouping them into closed loops, then forming cutting surfaces based on those loops. Although this algorithm is still in the research stage, it can be useful for automating some of the decomposition required for typical models. To automatically decompose a model, use the command

Cut Body <body_id_range> [Trace {on|off}] [Depth <cut_depth>]

If the Trace option is used, the algorithm prints progress information as decomposition progresses. The Depth option controls how many cuts are made before the algorithm returns; by default, the algorithm cuts the model wherever it can.

Automatic decomposition is used to decompose the model shown in Figure 4-2 left,, with the meshed results shown in Figure 4-2 right. In this case, automatic decomposition performs all but one of the required cuts.

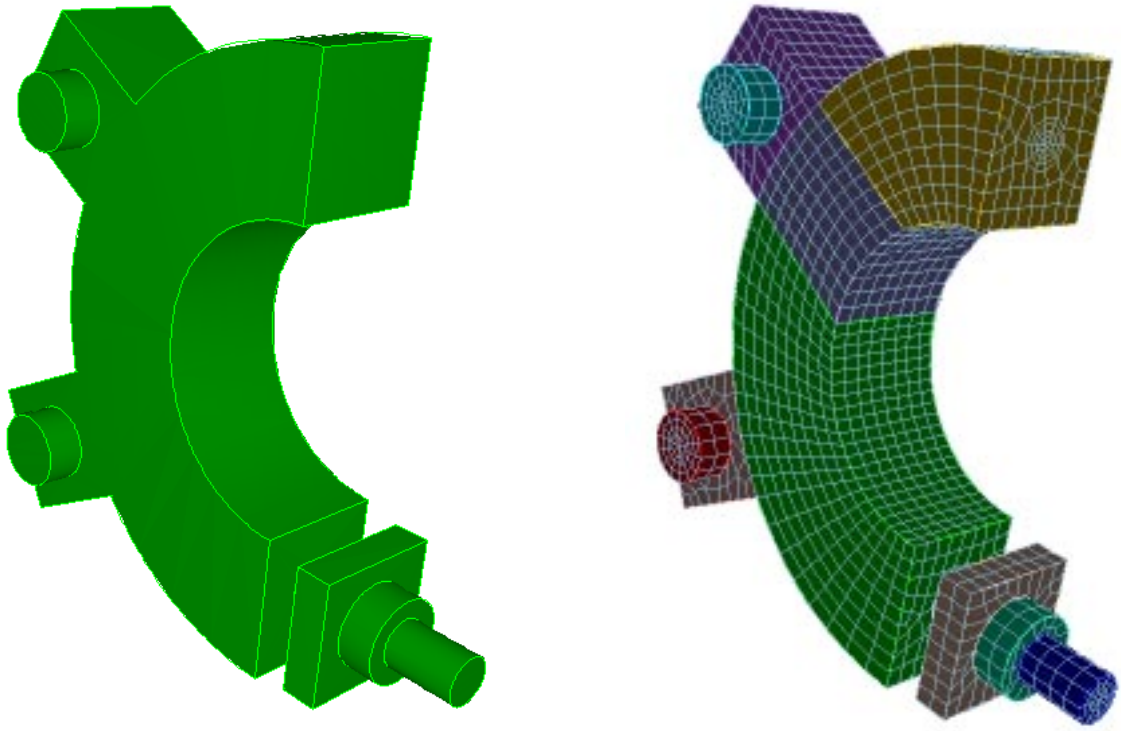


Figure 4-2: Automatic decomposition, plus one manual webcut, makes the model sweepable.

Geometry Merging

As stated in “Non-Manifold Topology” on page 60, geometry is created and imported in CUBIT in a manifold state, by default. The process of converting manifold to non-manifold geometry is referred to as “geometry merging”, since it involves merging multiple geometric entities into single ones.

The merging of two manifold surfaces into one non-manifold surface is depicted in Figure 4-3.

Figure 4-3: Merging two manifold surfaces into a single non-manifold surface.

It is clear that merging geometry results in some surfaces, curves and vertices being removed from the model. By default, entities with the lowest ID are retained.

Merging

There are several steps to the geometry merging algorithm in CUBIT; they are:

- Check lower order geometry, merge if possible
- Check topology of current entities
- Check geometry of current entities
- If both topology and geometry are alike, merge entities

Thus, in order for two entities to merge, the entities must correspond geometrically and topologically. The geometric correspondence usually comes from constructing the model that way. The topological correspondence can come from that process as well, but also can be accomplished in CUBIT using Imprinting (see “Imprint” on page 71.)

There are several options for merging geometry in CUBIT.

• *Merge geometry automatically*

Merge all [Vertex | Curve | Surface]

Merge Body <id_range>

All topological entities in the model or in the specified bodies are examined for geometric and topological correspondence, and are merged if they pass the test.

If a specific entity type is specified with the Merge all, only complete entities of that type are merged. For example, if Merge all surface is entered, only vertices which are part of corresponding surfaces being merged; vertices which correspond but which are not part of corresponding surfaces will not be merged. This command can be used to speed up the merging process for large models, but should be used with caution as it can hide problems with the geometry (see)

• *Test for merging in a specified group of geometry*

Merge [Vertex | Curve | Surface] <id_range>

All topological entities in the specfiied entity list, as well as lower order topology belonging to those entities, are examined for merging. This command can be used to prevent merging of entities which correspond and would otherwise be merged, e.g. slide surfaces.

- ***Force merge specified geometry entities***

Merge [Vertex | Curve | Surface] <id_range> Force

This command results in the specified entities being merged, whether they pass the geometric correspondence test or not. This command should only be used with caution and when merging otherwise fails; instances where this is required should be reported to the CUBIT development team.

Examining Merged Entities

There are several mechanisms for examining which enties have been merged. The most useful mechanism is assigning all merged or unmerged entities of a specified type to a group, and examining that group graphically. This process can be used to examine the outer shell of an assembly of volumes, for example to verify if all interior surfaces have been merged. To put all the merged or unmerged entities of a given type into a specified group, use the command:

Group {<'name'>|<id>} [Surface | Curve | Vertex] [Merged | Unmerged]

If the entity type is unspecified, surfaces will be assumed.

Entities can also be labelled in the graphics according to whether or not they have been merged. To turn merge labeling on for a specified entity type, use the command

Label {Vertex | Curve | Surface} Merge

Merge Tolerance

Geometric correspondence between entities is judged according to a specfied absolute numerical tolerance. The particular kind of spatial check depends on the type of entity. Vertices are compared by comparing their spatial position; curves are tested geometrically by testing points 1/3 and 2/3 down the curve in terms of parameter value; surfaces are tested at several pre-determined points on the surface. In all cases, spatial checks are done comparing a given position on one entity with the closest point on the other entity. This allows merging of entities which correspond spatially but which have different parameterizations.

The default absolute merge tolerance used in CUBIT is 5.0e-4. This means that points which are at least this close will pass the geometric correspondence test used for merging. The user may change this value using the following command:

Merge Tolerance <val>

If the user does not enter a value, the current merge tolerance value will be printed to the screen. There is no upper bound to the merge tolerance, although in experience there are few cases where the merge tolerance has needed to be adjusted upward. The lower bound on the tolerance, which is tied to the accuracy of the solid modeling engine in CUBIT, is 1e-6.

Using Geometry Merging to Verify Geometry

Geometry merging is often used to verify the correctness of an assembly of volumes. For example, groups of unmerged surfaces can be used to verify the outer shell of the assembly (see "Examining Merged Entities" on page 81.) There is other information that comes from the Merge all command that is useful for verifying geometry. In typical geometric models, vertices and curves which get merged will usually be part of surfaces containing them which get merged.

So, if a Merge all command is used and the command reports that vertices and curves have been merged, this is usually an indication of a problem with geometry. In particular, it is often a sign that there are overlapping bodies in the model. The second most common problem indicated by merging curves and vertices is that the merge tolerance is set too high for a given model. In any event, merged vertices and curves should be examined closely.

Geometry Groups

Groups provide a powerful capability for performing operations on multiple geometric entities with minimal input. They can also serve as a means for sorting geometric entities according to various criteria.

The command syntax to create or modify a group is:

group {id | “name”} add <list of topology entities>

For example, the command,

group “Exterior” add surface 1 to 2, curve 3 to 5

will create the group named **Exterior** consisting of the listed topological entities. Any of the commands that can be applied to the “regular” topological entities can also be applied to groups. For example, **mesh Exterior**, **list Exterior**, or **draw Exterior**. A topological entity can be removed from a group using the command:

group <id> remove <entity list>

When a group is meshed, CUBIT will automatically perform an interval matching on all surfaces in the group (including surfaces that are a part of volumes or bodies in the group).

There are several utilities in CUBIT which use groups as a means of visualizing output. These utilities are described elsewhere, but listed here for reference:

- Webcut results (see “Web Cutting” on page 73)
- Merged and unmerged entities (see “Examining Merged Entities” on page 81)
- Sweep groups (see “Web Cutting” on page 73)
- Interval matching (see “Web Cutting” on page 73)

Geometry Attributes

Each topological entity has attributes attached to it. These attributes specify aspects of the entity such as the color that entity is drawn in and the meshing scheme to be used when meshing that entity. This section describes those geometry attributes that are not described elsewhere in this manual.

Entity Names

Topological entities (including groups) are assigned integer identification numbers in CUBIT in ascending order, starting with 1. Each new entity created within CUBIT receives a unique id. However, topological entities can also be assigned names, and these names can be propagated explicitly by the user. A topological entity may have multiple names, but a particular name may

only refer to a single entity. The following command assigns names to topological entities in CUBIT:

{Group|Body|Volume|Surface|Curve|Vertex} Name '<entity_name>'

The name of each topological entity appears in the output of the **List** command. In addition, topological entities can be labeled with their names (see label command). A list of all names currently assigned and their corresponding entity type and id (optionally filtered by entity type) can be obtained with the command

list names [{group|body|volume|surface|curve|vertex|all}]

Note: In a merge operation, the names of the deleted entity will be appended to the names of the surviving entity.

Topological entities can be identified either by the entity type followed by an identification number or by a unique name. Such a name can be used anywhere that an entity type and id may be used. For example, if surface 3 is named CHAMFER1, the command "mesh CHAMFER1" has the same result as the command "mesh surface 3".

Each topological entity can optionally be given a unique default name when it is first created. The default name consists of the type of topological entity (body, volume, surface, curve, or vertex), followed by the ID number of the entity. For example, curve number 21 would have a default name, "curve21". Default names can be useful to reduce the amount of time spent redoing id's between CUBIT and ACIS versions. The command for setting the default names is:

set default names {on|off}

Persistent Attributes

Typical data assigned to topological entities during a meshing session might include intervals, mesh schemes, group assignments, etc. By default, most of this data is lost between CUBIT sessions, and must be restored using the original CUBIT commands. Using CUBIT's persistent attributes capability, some of this data can be saved with the solid model and restored automatically when the model is imported into CUBIT.

Attribute Behavior

In this context, attributes are defined as data associated directly with a particular geometry entity. In CUBIT's implementation of attributes, these data can occupy one of three "states" at any given time: they can exist only on the ACIS objects; they can exist in CUBIT's attribute objects; or they can be written to the appropriate data fields on CUBIT's geometry entities. Movement of data between these states is defined by the following behaviors:

- ***Read:*** read data from ACIS objects into CUBIT attribute objects
- ***Actuate*** :assign data from CUBIT attribute objects to CUBIT geometry entities
- ***Update :***assign data from CUBIT geometry entities to CUBIT attributes (opposite of Actuate)
- ***Write :***write data from CUBIT attribute objects to ACIS objects (opposite of Read)

By default, the Actuate and Update functions are not performed automatically, and must be requested by the user, either for specific geometric entities or for all entities for a given attribute type. The Read and Write functions are performed automatically, and are not normally controlled by the user.

Attribute Types

The attribute types currently implemented in CUBIT are shown in Table 4-2. There are also plans for implementing attributes based on block, nodeset, sideset, mesh scheme, and several other data.

Table 4-2: Attribute types currently implemented in CUBIT. All attributes are set to automatically read and write from and to ACIS model.

Attribute Type	Description	Default Actuate	Default Update
Composite VG	Information required to restore composite virtual geometry entities.	Manual	Manual
Genesis Entity	Genesis entities (blocks, nodesets, sidesets) to which an entity belongs.	Manual	Manual
Group	Groups to which the entity belongs.	Manual	Manual
Id	Id assigned to the entity in the current session.	Manual	Manual
Interval	Interval number, size and firmness.	Manual	Manual
Merge	Information on which other entities are merged with this one.	Manual	Manual
Mesh Container	Mesh owned by an entity.	Manual	Manual
Mesh Scheme	Mesh scheme and any data specific to the mesh scheme assigned to an entity.	Manual	Manual
Name	Entity name assigned by user or by default.	Auto	Auto
Partition VG	Information required to restore partition virtual geometry entities.	Manual	Manual
Relative Length	Relative length factor.	Manual	Manual
Smooth Scheme	Smooth scheme and any data specific to the smooth scheme assigned to an entity.	Manual	Manual
Vertex Type	Vertex type(s) assigned to a vertex.	Manual	Manual

Attribute Commands

The following commands are used to control attribute behavior in CUBIT.

{geom_list} Attribute {all | attribute_type} {actuate | remove | update | read | write}

Calls the function (actuate, update, etc.) for the designated attribute for the designated entity.

Set Attribute <attribute_type> Auto {actuate | update} {on | off}

Turn on or off the automatic actuation or updating for the given attribute.

List {Body | Volume | Surface | Curve | Vertex} <id_range> Attributes

List the attributes currently residing on the given entity.

Using CUBIT Attributes

A typical scenario for using CUBIT attributes would be as follows.

1. Construct geometry, merge, assign intervals, groups, etc. (i.e. normal CUBIT session)

2. Set the attribute option on using **Set Attribute on**.

3. Export acis file (see export acis command).

Subsequent runs:

1. **Set Attribute on.**

2. Import acis file (see import acis command); all attributes stored on model get actuated, either writing the data to the geometric entities (e.g. interval, name) or performing some action (e.g. merge, group).

Used in this manner, geometry attributes allow the user to store some data directly with the geometry, and have that data be assigned to the corresponding CUBIT objects without entering any additional commands.

▼ Exporting Geometry

Geometry can also be exported from within CUBIT to the ACIS SAT format. The SAT format can be used to exchange geometry between ACIS-compliant applications. The user can, optionally, specify which subset of bodies are to be exported. The command used to export geometry is:

Export Acis '<acis_sat_filename>' [Body <body_id_range>]

Note that the filename is enclosed in single or double quotes. If the **Body** keyword is not specified, then all the bodies in the model are saved. Note that the model is saved as manifold geometry, and will have that representation when imported back into CUBIT (see"Non-Manifold Topology" on page 60 and "Geometry Merging" on page 80.)

▼ New Geometry Commands

Tweaking Geometry

The tweaking commands modify models by moving, offsetting or replacing surfaces, while extending the adjoining surfaces to fill the resulting gaps. This is useful for eliminating gaps between components, simplifying geometry or changing the dimensions of an object.

Tweak Surface <id_range> Offset <value> [keep]

Tweak Surface <id_range> Move {Vertex|Curve|Surface|Volume|Body} <id>
Location {Vertex|Curve|Surface|Volume|Body} <id>
[Except [X][Y][Z]] [keep]

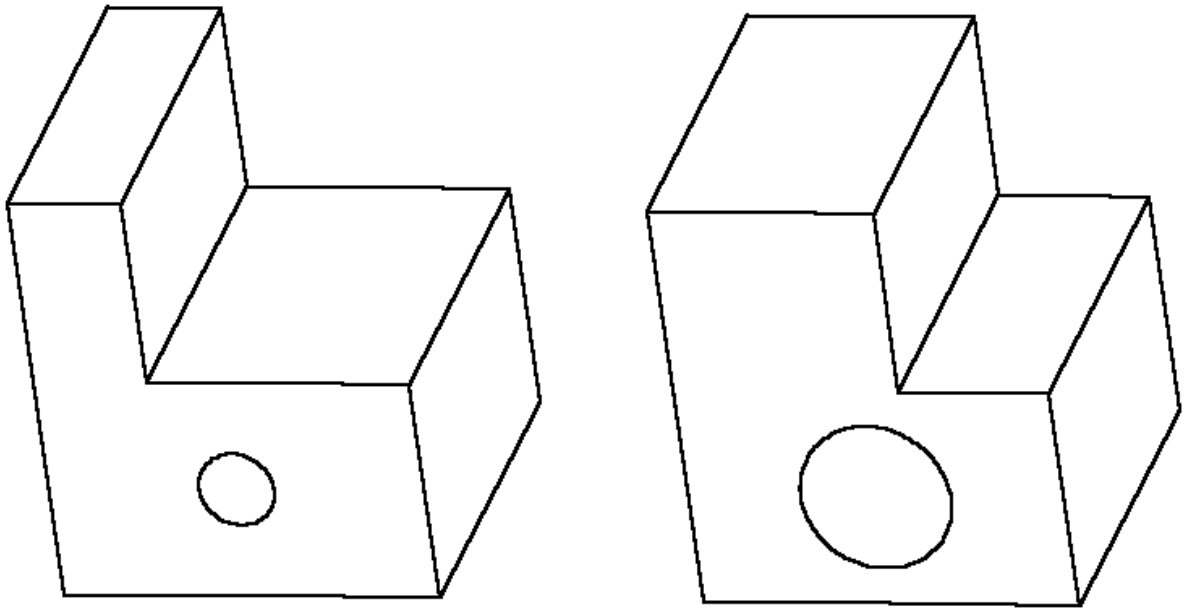
Tweak Surface <id_range> Move {Vertex|Curve|Surface|Volume|Body} <id>
Location <x_val> <y_val> <z_val> [Except [X][Y][Z]] [keep]

Tweak Surface <id_range> Move <dx_val> <dy_val> <dz_val> [keep]

Tweak Surface <id_range> Move Normal To Surface <id> Distance <val> [Except [X][Y][Z]] [keep]

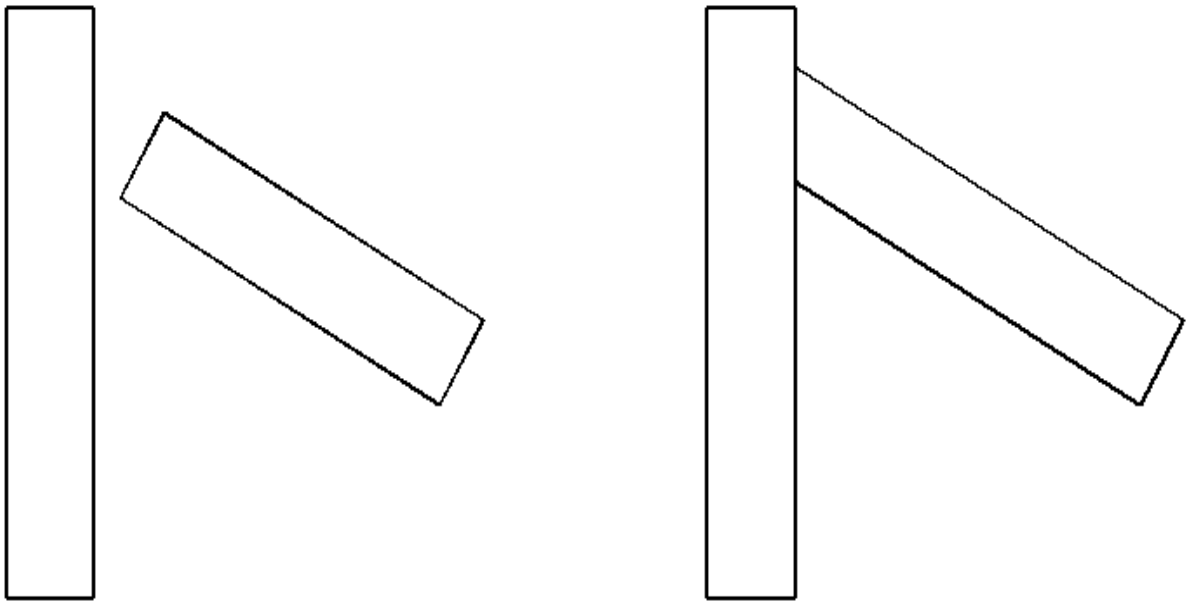
Tweak Surface <id_range> Replace [With] Surface <id> [keep] [reverse]

The first form offsets an existing set of surfaces and extends the attached surfaces to meet them. A positive offset value will offset the surface in the positive surface normal direction while a negative value will go the other way. Figure 1 shows a simple example of offsetting. Note that you can also offset whole groups of surfaces at once.



The next three forms of the command simply move the given surfaces along a vector direction. The direction can be specified either absolutely or relative to other geometry entities in the model (from entity centroid to location). Note that when moving a surface for tweak, the surface is moved and it and the adjoining surfaces are extended or trimmed to match up again. So, for example, moving a vertically oriented planar surface in the vertical direction will have no affect. In this example, if you move the surface 10 in the x and 5 in the y the effect will be to move it simply 10 in the x. You can also use these 3 forms of the command to move a protrusion around – just be sure to specify all of the surfaces on the protrusion for moving

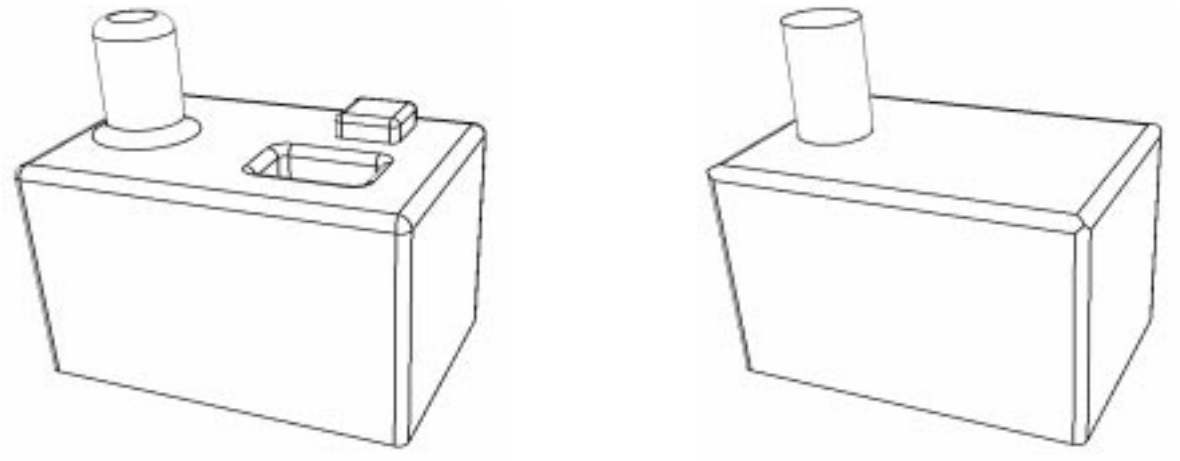
The last form of the command actually replaces the given surface with a copy of the new surface, then extends and trims surfaces to match up. This can be useful for closing gaps between components or performing more complicated modifications to models. Figure 2 shows a simple example.



Removing Surfaces

The remove surface command removes surfaces from bodies. By default, it attempts to extend the adjoining surfaces to fill the resultant gap. This is a useful way to remove fillets and rounds and other features such as bosses not needed for analysis. See Figure 3 for an example.

Remove Surface <id_range> [EXTEND|noextend] [keepsurface] [keep]



The *noextend* qualifier prevents the adjoining surfaces from being extended, leaving a gap in the body. This is sometimes useful for repairing bad geometry – the surface can be rebuilt with surface from curves or a net surface, etc.., then combined back onto the body.

Creating Vertices

The following commands have been added for creating vertices:

Create Vertex On Curve <curve_id>
{Fraction <f> | Distance <d> | Position <xval><yval><zval>
| Close_To Vertex <vertex_id> }
[From Vertex <vertex_id> (optional for 'Fraction' & 'Distance')]
[Color <color_name>]

Create Vertex AtArc Curve <id_list> [Color <color_name>]

Create Vertex AtIntersection Curve <id1> <id2> [bounded] [near] [Color <color_name>]

The first form is a general purpose command for creating a vertex on a curve. It allows the vertex to be created at a fractional distance along the curve, at an actual distance from one of the curves ends, or at the closest location to an xyz position or another vertex. You can preview the location first with the command **Draw Location On Curve**… (where the rest of the command is identical to the Create Vertex form).

The second form simply creates vertices at arc or circle centers. The last form creates vertices at the intersection of two curves. If the *bounded* qualifier is used, the vertices are limited to lie on the curves, otherwise the extensions of the curves are also used to calculate the intersections. The *near* option is only valid for straight lines, where the closest point on each curve is created if they do not actually intersect (resulting in two new vertices).

Creating Curves

The following command creates an arc either through 3 vertices or tangent to 3 curves. The *Full* qualifier will cause a complete circle to be created.

Create Curve Arc Three {Vertex|Curve} <id_list> [Full]

The following command creates an arc using the center of the arc and 2 points on the arc. The arc will always have a radius at a distance from the center to the first point, unless the *Radius* value is given. Again, the *Full* qualifier will cause a complete circle to be created.

*****Requires 3 Vertices - first is center, other two are on the arc*****
Create Curve Arc CenterEdge Vertex <id_list> [Radius <value>] [Full]

The following command will create a curve from a vertex onto a specified position along a curve. If none of the optional parameters are given, the location on the curve is calculated as using the shortest distance from the start vertex to the curve (i.e., the new curve will be normal to the existing curve).

*****Default = Normal to the Curve*****
Create Curve From Vertex <vertex_id> Onto Curve <curve_id>
[Fraction <f> | Distance <d> | Position <xval><yval><zval> |
Close_To Vertex <vertex_id>
[[From] Vertex <vertex_id> (optional for 'Fraction' & 'Distance')]]
[On Surface <surface_id>]

The following command creates curves offset at a specified distance from a planar chain of curves. The direction vector is only needed if a single straight curve is given. The offset curves are trimmed or extended so that no overlaps or gaps exist between them. If the curves need to be extended the extension type can be *Rounded* like arcs, *Extended* tangentially (the default -straight lines are extended as straight lines and arcs are extended as arcs), or extended *naturally.*

*****Direction is optional for offsets of individual straight curves only*****
Create Curve Offset Curve <id_list> Distance <val> [Direction <x> <y> <z>]
[Rounded|EXTENDED|Natural]

Trimming/Extending Curves

Curves can be *trimmed or extended* with the following command:

Trim Curve <id> AtIntersection {Curve|Vertex <id>} Keepside Vertex <id> [near]

The curve can be trimmed or extended where it intersects with another curve or at a vertex location. When trimming to another curve, the curves must physically intersect unless they both are straight lines in which case the *near* option is available. With the near option the closest intersection point is used to the other line – so it is possible to trim to a curve that lies in a different plane. When trimming to a vertex, if the vertex does not lie on the curve, it is projected to the closest location on the curve or an extension of the curve if possible.

The *Keepside* vertex is needed to determine which side of the curve to keep and which side to throw away. This vertex need not be one of the curve's vertices, nor need it lie on the curve. However, if it is not on the curve it will be projected to the curve and that location will determine which side of the curve to keep.

If the curve is part of a body or surface, it is simply copied first before trimming/extending. If it is a free curve a new curve is created and the old curve is removed. Figure 4 shows several examples of trimming/extending curves.

Creating Surfaces

The following commands create ***planar*** surfaces. The first passes a plane through 3 vertices, the second uses an existing plane, the third creates a plane normal to one of the global axes, and the fourth creates a plane normal to the tangent of a curve at a location along the curve. By default, the commands create the surface just large enough to intersect the bounding box of the entire model with minimum surface area. Optionally, you can give a list of bodies to intersect for this calculation. You can also extend the size of the surface by either a percentage distance or an absolute distance of the minimum area size. The plane can be previewed with the command **Draw Plane [with]…** (where the rest of the command is the same as that to create the surface).

Create Planar Surface [with] Plane Vertex <v1_id> [vertex] <v2_id> [vertex] <v3_id>
[intersecting] Body <id_range>] [extended percentage|absolute <val>]

Create Planar Surface [with] Plane Surface <surface_id>
[intersecting] Body <id_range>] [extended percentage|absolute <val>]

Create Planar Surface [with] Plane {xplane|yplane|zplane} [offset <val>]
[intersecting] Body <id_range>] [extended percentage|absolute <val>]

Create Planar Surface [with] Plane Normal To Curve <curve_id>
{fraction <f> | distance <d> | position <xval><yval><zval>
| close_to vertex <vertex_id>}
[[from] Vertex <vertex_id> (optional for 'fraction' & 'distance')]
[intersecting] Body <id_range>] [extended percentage|absolute <val>]

Net surfaces can be created with two different commands. A net surface passes through a set of curves in the u-direction and a set of curves in the v-direction (these u and v curves would looked like a mapped mesh). The first form of the command uses curves to create the net surface. The curves must pass within tolerance of each other to work. The second form uses a mapped mesh to create the surface. The mapped mesh can be of a single surface or a collection of mapped or submapped surfaces that form a logical rectangle. By default net surfaces are healed to take advantage of any possible internal simplification.

Create Surface Net U Curve <id_list> V Curve <id_list> [Tolerance <value>]
[HEAL|noheal]

Create Surface Net [From] [Mapped] Surface <id_list> [Tolerance <value>]
[HEAL|noheal]

A suggested geometry cleanup method is to use a virtual composite surface to map mesh a set of complicated surfaces then create a net surface from this mesh. Then the original surfaces can be removed with the *noextend* option and the new net surface combined back onto the body.

The following command creates surfaces ***offset*** from existing surfaces at the specified distance. The surfaces are not guaranteed to be extended or trimmed to share boundaries; however they are generally close.

Create Surface Offset [From] Surface <id_list> Distance <val>

The following command creates a ***skin*** surface from a list of curves. An example of a skin surface is to create a surface through a set of parallel lines.

Create Surface Skin Curve <id_list>

Creating Bodies

The following command creates a body offset from another body at the specified distance. The new surfaces are extended or trimmed appropriately. A positive distance results in a larger body; a negative distance in a smaller body.

Create Body Offset [from] Body <id_range> Distance <value>

This command allows the user to sweep a planar surface along a curve:

Sweep Surface <surface_id_range> Along Curve <curve_id> [draft_angle <degrees>] [draft_type <0 | 1 | 2>]

One of the ends of the curve must fall in the plane of the surface and the curve cannot be tangential to the surface. Sweep along curve also supports an additional draft type "2" which implies a "natural" extension of the corners from their curves.

Webcutting

Two new webcutting commands have been added.

Webcut {body|group} {<body_id_range>|all} [with] Plane Normal To Curve <curve_id> {fraction <f> | distance <d> | position <xval><yval><zval> | close_to vertex <vertex_id>} [[From] Vertex <vertex_id> (optional for 'fraction' & 'distance')] [NOIMPRINT|imprint][NOMERGE|merge][group_results]

Webcut {Body|Group} <id_range> [With] Loop [Curve] <id_range> NOIMPRINT|Imprint] [NOMERGE|Merge] [group_results]

The first command allows a user to specify an infinite cutting plane by specifying a location on a curve. The cutting plane is created such that it is normal to the curve tangent at the specified location.

The position on the curve can be specified as:

1) A fraction along the curve from the start of the curve, or optionally, from a specified vertex on the curve.
2) A distance along the curve from the start of the curve, or optionally, from a specified vertex on the curve.
3) An xyz position that is moved to the closest point on the given curve.
4) The position of a vertex that is moved to the closest point on the given curve.

The point on the curve can be previewed with the **Draw Location On Curve** command.

The second form cuts the body list with a temporary sheet body formed from the curve loop. This is the same sheet as would be created from the command **Create Surface Curve <id_list>.**

Imprinting

The following commands can be used for imprinting. A body can be imprinted with curves or vertices and surfaces can be imprinted with curves. It is useful to imprint bodies or surfaces with curves to eliminate mesh skew, generate more favorable surfaces for meshing, or create hardlines for paving. Imprinting with a vertex can be useful to split curves for better control of the mesh or create hardpoints for paving.

Imprint Body <body_id_range> [with] Curve <curve_id_range> [keep]

Imprint Body <body_id_range> [with] Vertex <vertex_id_range> [keep]

Imprint Surface <surface_id_range> [with] Curve <curve_id_range> [keep]

Validating Geometry

More rigorous checking can be accomplished with the validate geometry commands by specifying a higher check level. Use the following command to accomplish this:

set AcisOption Integer 'check_level' <integer>, where integer is one of the following:

10 = Fast error checks
20 = Level 10 checks plus slower error checks (default)
30 = Level 20 checks plus D-Cubed curve and surface checks
40 = Level 30 checks plus fast warning checks
50 = Level 40 checks plus slower warning checks
60 = Level 50 checks plus slow edge convexity change point checks
70 = Level 60 checks plus face/face intersection checks

You can also get more detailed output from the validate command with (the default is *off*):

set AcisOption Integer 'check_output' on

Note that some of the ids listed in the output of the validate command are currently meaningless. This will be corrected in a future release of CUBIT.

Geometry Accuracy

You can control the accuracy setting of the ACIS solid model geometry with the following command:

[set] Geometry Accuracy <value = 1e-6>

Some operations, like imprinting, can be more successful with a lower accuracy setting (i.e., 0.1 to 1e-5). However, it is not recommended to change this value. ***Be sure to set it back to 1e-6 before exporting the model or doing other operations as a higher setting can corrupt your geometry.***

Healing

Healing is an optional module that detects and fixes models in ACIS (CUBIT's core solid modeling kernel).

It is possible to create ACIS models that are not accurate enough for ACIS to process. This most often happens when geometry is created in some other modeling system and translated into an ACIS model. Such models may be imprecise due to the inherent numerical limitations of their parent systems, or due to limitations of data transfer through neutral file formats. This imprecision can also result when an ACIS model is created at a different tolerance from the current tolerance settings.

This imprecision leads to problems such as geometric errors in entities, gaps between entities, and the absence of connectivity information (topology). Since ACIS is a high precision modeler, it expects all entities to satisfy stringent data integrity checks for the proper functioning of its algorithms. Therefore, if such imprecise models must be processed by an ACIS based system, "healing" of such models is necessary to establish the desired precision and accuracy.

Bad geometry can cause boolean operations, such as imprinting and webcutting, to fail. However, it usually has little effect on meshing operations.

Analyzing Geometry

The following command analyzes the ACIS geometry and will indicate problems detected. Note that it is not necessary to analyze the geometry before healing; however, it can be useful to analyze first rather than healing unnecessarily. Also note that healer analysis can take a bit of time, depending on the complexity of the geometry and how bad the geometry is.

Healer Analyze Body <id_range> [logfile ['filename'] [display]]

The outputs include an estimate of the percentage of good geometry in each body. The optional logfile will include detailed information about the geometry analysis. By default CUBIT will also highlight the bad geometry in the graphics and give a printed summary indicating which entities are "bad".

```
Percentage good geometry in Body 9: 98%

HEALER ANALYSIS SUMMARY:
------------------------
Analyzed 1 Body: 9
Found 2 bad Vertices: 51, 52
Found 3 bad Curves: 76, 77, 80
Found 2 bad CoEdges. The Curves are: 76
Found 1 Bodies with problems: 9
Journaled Command: healer analyze body 9
```

If you try to do a webcut or boolean operation through the indicated entities, it will likely fail. The actual problem is not indicated here (future versions of CUBIT may include more detailed information), but common problems include vertices that do not lie on curves, and curves that do not lie sufficiently close to surfaces. The *validate geometry* commands work independently of the healer and give more detailed information.

You can control the outputs from the healer with the following commands:

Healer Set OnShow {highlight|draw|none}

Healer Set OnShow {badvertices|badcurves|badcoedges|badbodies|all} {On|Off}

Healer Set OnShow Summary {On|Off}

These settings allow you to highlight, draw or ignore the bad entities in the graphics. You can control which entity

types to display, as well as whether or not to show the printed summary at the end of analysis.

After you have analyzed the geometry (which can take some time), you can show the bad geometry again with the "show" command. This command simply uses cached data (healing attributes – see the next section) from the previous analysis.

Healer Show Body <id_list>

Healing Attributes

Once the geometry is analyzed, the results are stored as attributes on the solid model - this allows you to use the "show" command to quickly display the bad geometry again. The results attributes are automatically removed when the geometry is exported or any boolean operations are performed. They can also be explicitly removed with the command:

Healer CleanAtt Body <id_range>

You can force the results to be removed immediately after each analyze operation with the "CleanAtt" setting (this can save a little memory):

Healer Set CleanAtt {On|Off}

AutoHealing

Healing is an extremely complex process. The general steps to healing are:

1. Preprocess – trim overhanging surfaces and clean topology (remove small curves and surfaces).
2. Simplify – converts splines to analytic representations, if possible.
3. Stitch – geometry cleanup and and stitching loose surfaces together to form bodies.
4. Geometry Build – repairing and building geometry to correct gaps in the model.
5. Post-Process – calculating pcurves and further repairing bad geometry.
6. Make Tolerant Curves & Vertices – a last optional step that allows special handling of unhealed entities for booleans – allowing inaccurate geometry to be tolerated.

Autohealing makes these steps pushbutton with the following command:

Healer Autoheal Body <id_range> [rebuild] [keep] [maketolerant]
[logfile ['logfilename'] [display]]

The "rebuild" option actually unhooks each surface, heals it individually, then stitches all the surfaces back together and heals again. In some cases this can more effectively fixup the body, although it is much more computationally intensive and is not recommended unless healing is unsuccessful.

The "maketolerant" option will make the edges *tolerant* if ACIS is unable to heal them. This can result in successful booleans even if the body cannot be fully healed – ACIS can then sometimes "tolerate" the bad geomety. Note that the *healer analyze* command will still show these curves as "bad", even though they are tolerant. The *validate geometry* commands however take this into consideration.

The output from the autoheal command will include an analysis of the geometry – this output is controlled with the same "onshow" settings described earlier in the analysis section. The optional logfile will include detailed information about the healing process.

Later versions of CUBIT will allow you to individually control each step and the internal tolerances used within. At this time only the commands documented here are supported.

What if Healing is Unsuccessful?

The healing module is under continual development and is improving with every release. However, there will probably always be situations where healing is unable to fully correct the geometry. This might be okay, as meshing is rarely affected by the small inaccuracies healing deals with. However, boolean operations on the geometry (webcut, unite, etc..) can fail if the bad geometry must be processed by the operation (i.e., a webcut must cut through a bad curve or vertex).

Here are some possible methods to fix this bad geometry:

1. Return to the source of the geometry (i.e., Pro/ENGINEER) and increase the accuracy. Re-export the geometry.
2. Heal again using the *rebuild* option.
3. Heal again using the *make tolerant* option.
4. Remove the offending surface from the body (using *remove surface <id> noextend*). Then construct new surfaces from existing curves and combine the body back together.
5. Composite the surfaces over the bad area, mesh and create a net surface from the composite, remove the bad surfaces and combine.
6. Export the geometry as IGES, import the IGES file into a new model and look for double surfaces or surfaces that show up at odd angles. Delete and recreate surfaces as needed and combine the surfaces back together into a body.

Contact the development team if you need further help with fixing bad geometry.

Surface Overlap

The surface overlap capability finds surfaces that *overlap* each other, with the capability to specify a distance and angle range between them. This is useful for debugging geometry imprinting and merging problems, as well as for finding gaps in large assembly models (oftentimes gaps between components are left in assemblies to accommodate for tolerance fitup or welds – in general these gaps need to be filled for meshing). Finding non-imprinted, non-merged or non-touching surfaces manually can be tedious and time consuming. This tool can automate the process. The command is:

Find [Surface] Overlap [Body <id_list>]

If you do not specify a body list it will work on all the bodies in the model. The command will not check the surfaces within a given body – rather, it only checks surfaces between bodies. You can optionally limit the search to several bodies by listing their ids.

Facetted Representation

This command works entirely off of the facetted surface representation of the model (the facetted representation is what you see in a shaded view in the graphics). There are inherent advantages and disadvantages with this method. The biggest advantage is avoidance of closest-point calculations with NURBS based geometry, which tends to be slow. This method also eliminates possible problems with unhealed ACIS geometry. The disadvantage is working with a less accurate (i.e., facetted) representation of the geometry. To circumvent problems with this facetted geometry, various settings can be used to control the algorithm.

You might consider generating a more accurate facetted representation of the model before using this command. This can be done with the **Graphics Tolerance {Angle|Distance} <value>** command, followed by an **Update Hack**. This will rebuild the entire graphics tree using the new tolerance values you specified. The angular tolerance indicates the maximum angle between normals of adjacent surface facets. The default angular tolerance is 15° - consider using a value of 5°. The distance tolerance means the maximum actual distance between the generated facets and the surface. This value is by default ignored by the facetter – consider specifying a reasonable value here for more accurate results.

Find Overlap Settings

You can list out the settings that find overlap uses with:

Find [Surface] Overlap Settings

These settings can be controlled with the following commands:

set Overlap {Minimum|Maximum} Gap <value=0.0 to 0.01>
set Overlap {Minimum|Maximum} Angle <value=0.0 to 5.0>
set Overlap Normal {ANY|opposite|same}
set Overlap Tolerance <value=0.001>
set Overlap Group {ON|off}
set Overlap List {ON|off}
set Overlap Display {ON|off}

Here is an explanation of each setting:

Gap – Minimum/Maximum – the algorithm will search for surfaces that are within a distance from the minimum to maximum specified. The default range is 0 to 0.01. Since we are working with facets, testing has shown that this is about right for most situations, if you are looking for coincident surfaces. If you are looking for gaps, rather than touching surfaces, you can give a range such as 3.95 to 5.05.

Angle – Minimum/Maximum – the algorithm will search for surfaces that are within this angle range of each other. The default range is 0.0 to 5.0 degrees. Testing has shown that this range works well for most models. It is usually

necessary to have a range up to 5.0 degrees even if you are looking for coincident surfaces because of the different types of facetting that can occur on curvy type surfaces. For example, for the case of a shaft in a hole, the facets of the shaft usually won't be coincident with the facets of the hole, but may be offset by a certain distance circumferentially with each other. The 5 degree max angle range will account for this. If you find that the algorithm is not finding coincident surfaces when it should, you can increase the upper range of this value. Note that this parameter is useful also for finding plates coming together at an angle.

Normal – this setting determines whether to search for surfaces whose normals point towards each other (same), away from each other (opposite) or either (any). The default is ANY, but it may be useful to limit this search to *opposite*, as this would be the usual case for most finds.

Tolerance – two individual facets must overlap by more than this area for a match to be found. Consider the two cylindrical curves at the interface of the shaft and the block in Figure 5. Note that some of the facets actually overlap, even though the curves will analytically be coincident. You can filter out false matches by increasing the overlap tolerance area.

Group – the surface pairs found can optionally be placed into a group. The name of the group defaults to "overlap_surfaces".

List – by default the command lists out each overlapping pair - you can turn this off.

Display – by default the command clears the graphics and displays each overlapping pair – you can turn this off.

▼ Model Import/Export

Importing/Exporting ACIS Files

The import/export capability of ACIS files has been enhanced to support the binary format (.sab) and to include free entities (vertices, curves and surfaces) in the file. The import/export commands are:

Import Acis '<acis_filename>' [no_bodies][no_surfaces][no_curves][no_vertices] [binary|ascii] [current]

Export Acis [Debug] 'filename' [Body <id_list> Surface <id_list> Curve <id_list> Vertex <id_list>] [binary|ascii] [current] [overwrite]

When importing or exporting, the filename extension will determine the default file type, be it ascii or binary. At .sat extension will default to ascii; a .sab extension will default to binary. If you use a different file extension you can

specify the type with the [binary|ascii] option (with an unsupported extension exporting will default to ascii but importing requires the type to be specified). Binary files can be significantly faster but are not guaranteed to be upward compatible nor cross-platform compatible (although testing has determined compatibility between NT and HP/UX). Please archive your models in ascii format.

The current option will set the default filename for autosave (cntrl-S or File->Save (auto inc)) to the imported filename. Also, the filename is then set in the window titlebar.

When importing, you can turn off the import of certain entity types – be it bodies, surfaces, curves or vertices.

When exporting, you can now export individual surfaces, curves and vertices in addition to bodies. The software will check to see if the file exists already – if it does a dialogue will popup requiring you to confirm an overwrite. The journaled command will include the [overwrite] option.

Importing/Exporting STEP Files

The ACIS STEP translator has been added to CUBIT. This provides bi-directional functionality for data translation between ACIS and the file format standards STEP AP203 and STEP AP214.

STEP AP203 is an international standard which defines a neutral file format for representation of configuration control design data for a product. STEP AP214 is an international standard which defines a neutral file format for representatation of automotive design data. It is recommended to use AP214 for exchange of geometry information with CUBIT.

The commands used to import and export a STEP file are:

Import Step '<step_filename>' [no_bodies][no_surfaces][no_curves][no_vertices]
[HEAL|noheal] [logfile ['filename'] [display]]

Export Step 'filename' [Body <id_list> Surface <id_list> Curve <id_list>
Vertex <id_list>] [logfile ['filename'] [display]] [overwrite]

As with ACIS file import, you can control which types of entities to read. By default, bodies are automatically healed when imported - if this causes problems, you can disable this option. You can also optionally request a detailed logfile of the conversion process and display it in a text editor.

As with ACIS file export, you can specify which individual entities to export. Again, you can produce a logfile showing the conversion status.

To export a STEP file from Pro/ENGINEER, from the Export STEP Dialog, Press Options...
In step_config.pro add: STEP_EXPORT_FORMAT AP214_CD. Also be sure your export option is set to *Solids*. If the geometry has problems in CUBIT, you may need to increase the geometry accuracy in Pro/ENGINEER.

Importing/Exporting IGES Files

The ACIS IGES translator has been added to CUBIT. This provides bi-directional functionality for data translation between ACIS and the IGES (Initial Graphics Exchange Specification) format.

The commands to import/export IGES files are:

Import Iges '<iges_filename>' [no_bodies][no_surfaces][no_curves][no_vertices]
[nofreesurfaces] [logfile ['filename'] [display]]

Export Iges 'filename' [Body <id_list> Surface <id_list> Curve <id_list>
Vertex <id_list>] [logfile ['filename'] [display]] [overwrite]

The options here work similar to those for STEP, except on import the *nofreesurfaces* option will automatically convert free surfaces to bodies. By default this option is off.

This translator supports Manifold Solid B-rep Objects (MSBO) as well as Trimmed Surface Objects. By default, MSBO objects (i.e., bodies) will be converted to trimmed surfaces. If you want to support MSBO objects during import/export, use this command (the default is *off*):

set AcisOption Integer 'iges_proc_msbo' On

You can add this to your .cubit file so it is turned on during each session of CUBIT.

▼ Groups

The capability to store mesh entities in groups has been added to CUBIT. Groups can also now be deleted. The ability to propagate hexes and store them in groups has been added. In addition, element quality groups can now be created.

Add/Remove/Xor/Delete/Cleanout

The capability to add, remove, and xor mesh entities in the group command is possible. The commands are basically the same as for adding, removing , or xoring a geometric entity for groups.

Group ['name' | <id>] Add {hex|face|edge|node <id_list>}

Group ['name' | <id>] Remove {hex|face|edge|node <id_list>}

Group ['name' | <id>] Xor {hex|face|edge|node <id_list>}

Xor means if an entity is already in the group, the command will delete this entity from the group. If it is not in the group, the entity is then added to the group.

Groups can be deleted with the following command:

Delete Group <id range> [propagate]

The option *propagate* will delete the group specified and all of its contained groups recursively.

You can remove all of the entities in a group via the *cleanout* command:

Group <group_id_range> Cleanout [geometry|mesh] [propagate]

By default all entities will be removed – optionally you can cleanout just geometry or mesh entities. As in delete, the *propagate* option will cleanout the group specified and all of its contained groups recursively.

Groups in Graphics

When groups are created, they are now added to segments in the graphics tree. This means that you now have the ability to select a group graphically with the mouse, just like any other CUBIT entity. This can be useful in large models with lots of groups.

When displaying a group containing hexes, only the outside skin of the hexes will be displayed.

Propagated Hex Groups

The ability to propagate hexes given a starting face or surface and store them in groups has been added. There are various forms of this command, best illustrated by example.

Note: the examples below are based on first executing these commands:

brick width 10
volume 1 size 1
mesh volume 1

Starting on a Face

When starting on a face, the propagation method can end at a surface, end at a face or can end after the number of times the user specifies.

Ending at a surface:**Group ['name' | <id>] Add Propagate Face <id range> End Surface <id>**
example: *group 2 add propagate face 1 11 21 end surface 2*
result: Group 2 will be created containing 30 propagated hexes (10 layers of 3 hexes)

Ending at a face: **Group ['name' | <id>] Add Propagate Face <id> End Face <id>**
example: *group 2 add propagate face 1 end face 1721*
result: Group 2 will be created containing 5 propagated hexes (5 layers of 1 hex)

Note: Ending at a face requires starting at one face at one time, but ending at surface allows multiple start faces

Number of Times:**Group ['name' | <id>] Add Propagate Face <id range> Times <number>**
example: *group 2 add propagate face 2 times 4*
result: Group 2 will be created containing 4 propagated hexes (4 layers of 1 hex)

Both methods, ending at surface, end at a face or number of times, can be used with the "multiple" option which will create a grandparent (top-level), parent (mid-level, contained within the grandparent) and child (bottom level, contained within the parent) groups. The child groups will contain each hex layer (specified number of layers per child group), all organized into a single parent group, which is organized underneath the group ID given to the command. Subsequent propagation commands could then be executed adding to the grandparent group, but creating a new parent and child groups. This way multiple propagation “sets” can be stored in one grandparent group, if desired.

Ending at a surface:**Group ['name' | <id>] Add Propagate Face <id> End Surface <id>**
with multiple **Multiple <number>**
example: *group 2 add propagate face 1 end surface 2 multiple 1*
result: Ten groups will be created and stored with their respective ids, one for each layer of hexes. These groups will be stored in the parent group, Group 3, and Group 3 will be stored in the grand parent group, Group 2. A subsequent propagation command could be executed adding to group 2 (the grandparent), which would create a single group contained in group 2 (the parent), containing the hex layer groups (the children).

Ending at a face: **Group ['name' | <id>] Add Propagate Face <id> End Surface <id>**
with multiple **Multiple <number>**
example: *group 2 add propagate face 1 end face 1721 multiple 1*

result: 5 groups will be created and stored with their respective ids, one for each layer of hexes. These groups will be stored in the parent group, Group 3, and Group 3 will be stored in the grand parent group, Group 2. A subsequent propagation command could be executed adding to group 2 (the grandparent), which would create a single group contained in group 2 (the parent), containing the hex layer groups (the children).

If the end surface or end face is ambiguous, a node direction can be specified to direct the propagation. When specify the node direction, the node has to be picked such that when the hexes are propagated, the picked node lies in these propagated hexes. If that node is never reached while propagating, the direction is not found and zero hexes will be included in the specified group.

Ending at a face: **Group ['name' | <id>] Add Propagate Face <id> End Face <id>**
with direction **Direction Node <id>**
example: *group 2 add propagate face 1721 end face 1 direction node334*
result: group 2 will be created containing 6 hexes

Ending at a surface:**Group ['name' | <id>] Add Propagate Face <id range>**
with direction **End Surface <id> Direction Node <id>**
example: *group 2 add propagate face 1 end surface 2 direction node 334*
result: group 2 will be created containing 10 hexes

Note: The direction command and the multiple command can be used together (i.e *group 2 add propagate face 1721 end face 1 multiple 2 direction node 334*)

Number of Times:**Group ['name' | <id>] Add Propagate Face <id> Times <number>**
with multiple **Multiple <number>**
example: *group 2 add propagate face 1 times 10 multiple 5*
result: Two groups will be created and stored with their respective ids, these two groups will be stored in the parent group, Group 3, and Group 3 will be stored in the grand parent group, Group 2.

If number of times is specified and the direction is ambiguous, a surface direction or a node direction can be specified to direct the propagation. The node direction has the same condition as when ending at a surface or face and that is it must lie in the propagated hexes.

Number of Times:**Group ['name' | <id>] Add Propagate Face <id> Times <number>**
with direction **Direction [surface <id> | node <id>]**
example: *group 2 add propagate face 1721 times 4 direction surface 2*
example: *group 2 add propagate face 1721 times 4 direction node 334*
result: group 2 will be created contained 4 hexes

Note: The direction command and the multiple command can be used together. (i.e. *group 2 add propagate face 1721 times 4 multiple 2 direction surface 1*)

Starting on a Surface

Starting on a surface can end at a surface or can end after the number of times the user specifies.

Ending at a surface:**Group ['name' | <id>] Add Propagate Surface <id> End Surface <id>**
example: *group 2 add propagate surface 1 end surface 2*
result: group 2 will be created containing 1000 hexes

Number of Times:**Group ['name' | <id>] Add Propagate Surface <id> Times <number>**
example: *group 2 add propagate surface 1 times 4*
result: group 2 will be created containing 400 hexes

Both methods, ending at surface or number of times, can be used with the "multiple" option which will create several

groups depending upon the multiple number specified.

Ending at a surface:**Group ['name' | <id>] Add Propagate Surface <id> End Surface <id>**
with multiple **Multiple <number>**
example: *group 2 add propagate surface 1 end surface 2 multiple 2*
result: Five groups will be created and stored with their respective ids of multiple 2, these groups will be stored in the parent group, Group 3, and Group 3 will be stored in the grand parent group, Group 2.

Number of Times:**Group ['name' | <id>] Add Propagate Surface <id> Times <number>**
with multiple **Multiple <number>**
example: *group 2 add propagate sur face 1 times 10 multiple 5*
result: Two groups will be created and stored with their respective ids of multiple 5, these two groups will be stored in the parent group, Group 3, and Group 3 will be stored in the grand parent group, Group 2.

If number of times is specified and the direction is ambiguous, the surface direction or the node direction can be specified to direct the propagation. If the end surface is specified, only a node direction can be specified to direct the propagation. When specifying the node direction, the node has to be picked such that when the hexes are propagated, the picked node lies in these propagated hexes. If that node is never reached while propagating, the direction is not found and zero hexes will be included in the specified group.

Note: for the examples below, the result can be seen by executing these commands:

brick x 10
vol 1 size 1
brick width 10
body 2 move 10
volume all size 1
merge all
mesh volume all

Number of Times:**Group ['name' | <id>] Add Propagate Surface <id> Times <number>**
with direction **Direction [surface <id> | node <id>]**
example: *group 2 add propagate surface 6 times 4 direction surface 4*
example: *group 2 add propagate surface 6 times 4 direction node 1530*
result: group 2 will be created containing 400 hexes

Ending at a surface:**Group ['name' | <id>] Add Propagate Surface <id> Times <number>**
with direction **Direction Node <id>**
example: *group 2 add propagate surface 6 end surface 12 direction node 1530*
result: group 2 will be created containing 400 hexes

Note: The direction command and the multiple command can be combined (i.e. *group 2 add propagate surface 6 times 4 multiple 2 direction node 1530*)

Propagated Group Naming Convention

A special naming convention can be used for the propagated groups, best described by an example. The following command will create a hierarchy of logically named groups, as follows.

group 'W1P1T1' add propagate surf 1 end surf 2 multiple 1

The hierarchy looks like this:

W1
    W1P1
        W1P1T1

W1P1T2
W1P1T3
…
W1P1T10

Where W1P1 is contained within W1, and W1P1T1, W1P1T2, etc.. are contained within W1P1.

The software simply looks for numerical numbers in the group name and parses out the correct grandparent, parent and child names from the substrings. There must be exactly 3 substrings in the group name, each ending with an integer for the command to work properly.

A subsequent command:

group 'W1P2T1' add propagate surf 3 end surf 5 multiple 1

will add a parent group to W1, called W1P2, and the subsequent child groups:

W1
- W1P1
  - W1P1T1
  - W1P1T2
  - W1P1T3
  - …
  - W1P1T10
- W1P2
  - W1P2T1
  - W1P2T2
  - W1P2T3
  - …
  - W1P2T10

Quality Groups

Groups can also be formed from the hexes or faces obtained from the quality command. Each group formed using quality can be drawn with its associated quality characteristics {i.e jacobian low .2 high .3} automatically.

command: **group ['name'|<id>] add quality {volume|surface|group|hex|face} <id range> <metric name> [low <value> | bottom <value> | top <number> | bottom <number> | malformed]**

example:group 2 add quality volume 1 jacobian

result: (if the meshed brick from the first note in the section **Propagated Hex Groups** is used)
Group 2 will be created and it will contain 1000 hexes with quality characteristics.
If the group is drawn, its quality characteristics will be displayed automatically.

Entity Filtering

CUBIT entity filtering provides the user with the capability to quickly select the entities needed and to parse out those that match certain entity characteristics. In general, think of filtering as starting with a list of entities, than passing that list through a filter or set of filters which removes entities not meeting the filter criteria.

In general there are two modes to filtering:

1. **Execute a filter immediately**. This allows you to list, draw, highlight or select entities that meet a certain attribute criteria. For example, draw all the surfaces with scheme pave or draw all elements with a certain quality characteristic.

2. **Register a filter which is used in further operations**. Registered filters can be used to filter entities from the mouse pick list or keyed-in entity lists. Filters can be chained together (in a boolean AND/OR mode) to result in entities meeting a certain set of criteria.

Executing Filters

There are a large variety of filter types that can be executed immediately. It is generally recommended that the filters be run from the GUI, where you can see all of the types available. An example of the syntax is:

Execute [filter] {Volume|Surface|Curve} Mesh_Scheme <mesh_scheme>
[INCLUDE|Exclude] [on {Volume|Surface|Curve} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

The default is to run on all entities of the specified type; optionally you can give an entity list (i.e., get all paved surfaces on volumes 1 to 3). If you use the INCLUDE default, the filter will return all entities matching the input. If the EXCLUDE option is used, all entities other than the input will be returned (i.e., all surfaces not paved). The Execute command will always result in a listing of the entities – you can also draw or highlight them in the graphics. If drawing, if you use the Add qualifier they will be added to the display; otherwise the graphics are cleared first then they are drawn. The Select option will select the entities in the GUI (i.e., they will be current on the right mouse button and copied into the current selection list, if it matches the type of entities being filtered). You can also place the filtered entities into a group.

Quality Filter Commands

The filters support quality ranges of hexes or faces. Here is an example that draws the 10 hexes with the worst jacobian values in the model:

execute filter hex quality_range Jacobian bottom 10 include on hex all draw

This command will draw all the unshared element edges in the model, useful for finding cracks in a mesh:

execute filter edge owned_hexes equal_to 1 include on edge all draw

Other similar filters exist.

Execute Filter Commands

Following is a list of currently supported execute filter commands:

Execute [filter] Group Type <group_type>
[INCLUDE|exclude] [on Group <id_range>] [Draw|Highlight]
[into group {id|'name'}] [select] [add]

Execute [filter] {Group|Body|Volume|Surface|Curve|Vertex} Is_Meshed
[INCLUDE|exclude] [on {Group|Body|Volume|Surface|Curve|Vertex}] <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Group|Body|Volume|Surface|Curve|Vertex} Is_Visible
[INCLUDE|exclude] [on {Group|Body|Volume|Surface|Curve|Vertex} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Group Ref_Entities_Only
[INCLUDE|exclude] [on Group <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Group Mesh_Entities_Only
[INCLUDE|exclude] [on Group <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Group Is_Empty
[INCLUDE|exclude] [on Group <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Body Volume {greater_than|greater_than_equal|less_than|equal_to|
less_than_equal <value>} [INCLUDE|exclude] [on Volume <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Body|Volume|Surface|Curve} Interval_Number
{greater_than|less_than|equal_to <value>}
[INCLUDE|exclude] [on {Body|Volume|Surface|Curve} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Body Is_Sheet
[INCLUDE|exclude] [on Body <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Body Contained_Volumes Equal_To <number>
[INCLUDE|exclude] [on Body <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Body|Volume} Contained_Surfaces Equal_To <number>
[INCLUDE|exclude] [on {Body|Volume} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Body|Volume|Surface} Contained_Curves Equal_To <number>
[INCLUDE|exclude] [on {Body|Volume|Surfaces} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Body|Volume|Surface|Curve} Contained_Vertices Equal_To <number>
[INCLUDE|exclude] [on {Body|Volume|Surface|Curve} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Volume Range {greater_than|greater_than_equal|less_than|equal_to|
less_than_equal <value>} [INCLUDE|exclude] [on Volume <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Volume|Surface|Curve} Mesh_Scheme <mesh_scheme>
[INCLUDE|exclude] [on {Volume|Surface|Curve} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Volume|Surface} Smooth_Scheme <mesh_scheme>
[INCLUDE|exclude] [on {Volume|Surface} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Surface Area {greater_than|greater_than_equal|less_than|equal_to|
less_than_equal <value>} [INCLUDE|exclude] [on Surface <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Surface Is_Source
[INCLUDE|exclude] [on Surface <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Surface Is_Target
[INCLUDE|exclude] [on Surface <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Surface Is_Linking
[INCLUDE|exclude] [on Surface <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Surface|Curve} Is_Periodic
[INCLUDE|exclude] [on {Surface|Curve} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Surface Attached_Volumes Equal_To <number>
[INCLUDE|exclude] [on Surface <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Surface|Curve|Vertex} Merge_Partners
[Equal_To|Less_Than|Greater_Than <number>]
[INCLUDE|exclude] [on {Surface|Curve|Vertex} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Hex|Face} Quality_Range <metric_name>
{Low <double>|High <double>| Top <integer>| Bottom <integer>}
[INCLUDE|exclude] [on Hex|Face <id_range>] [Draw|Highlight]
[into group {id|'name'}] [select] [add]

Execute [filter] {Surface|Curve} Geometry_Type <geometry_type>
[INCLUDE|exclude] [on {Surface|Curve} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Surface Sizing_Function <sizing_function_type>
[INCLUDE|exclude] [on Surface <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Curve|Edge}
Length {greater_than|greater_than_equal|less_than|equal_to|
less_than_equal <value>} [INCLUDE|exclude] [on {Curve|Edge} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Curve Attached_Surfaces Equal_To <number>
[INCLUDE|exclude] [on Curve <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Curve Interval_Setting {Equal_To|Less_Than|Greater_Than <number>}
[INCLUDE|exclude] [on Curve <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Curve Is_Tolerant
[INCLUDE|exclude] [on Curve <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Vertex Attached_Curves Equal_To <number>
[INCLUDE|exclude] [on Vertex <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Body|Volume|Surface|Curve|Vertex|Hex|Face|Edge|Node}
Color <type>
[INCLUDE|exclude]
[on {Body|Volume|Surface|Curve|Vertex|Hex|Face|Edge|Node} <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Group|Body|Volume|Surface|Curve|Vertex|Hex|Face|Edge|Node}
Related_To
[on {Body|Volume|Node|Edge|Face} <id_range>]
[INCLUDE|exclude] [Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Hex Full_Hex [INCLUDE|exclude] [on Hex <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Hex Node_Hex [INCLUDE|exclude] [on Hex <id_range>]
[Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] {Face|Edge} Owned_Hexes Equal_To <number> [INCLUDE|exclude]
[on {Face|Edge} <id_range>] [Draw|Highlight] [into group {id|'name'}]
[select] [add]

Execute [filter] Face Free_Faces [INCLUDE|exclude]
[on Face <id_range>] [Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Edge Owned_Faces Equal_To <number> [INCLUDE|exclude]
[on Edge <id_range>] [Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Edge Free_Edges [INCLUDE|exclude]
[on Edge <id_range>] [Draw|Highlight] [into group {id|'name'}] [select] [add]

Execute [filter] Node Position_Fixed [INCLUDE|exclude]
[on Node <id_range>] [Draw|Highlight] [into group {id|'name'}] [select] [add]

Registering Filters

When filters are registered they can be used in conjunction with mouse picking and entity parsing. By default, the entities found during a mouse pick list will be passed through any registered filters. If any entities are removed during this process a warning message will be echoed to the output window. You can control whether filtering occurs during mouse selection with this command:

set Pick Filter [ON|off]

In GUI dialogue entity list fields, the registered filters will always filter the input list. On the command line you can cause draw,highlight and list commands to use the filters with the following setting:

set Parse Filter [on|OFF]

The draw, highlight and list commands can be override this setting by the options [filter | no filter].

There currently is not a way to filter every input list on the command line nor turn off the registered filters in GUI dialogue input lists.

Some of the GUI pages register filters internally – this can help the user to avoid unnecessary picks. For example, when webcutting with a plane normal to a curve, the end of the curve needs to be picked. Thus, after the curve is selected a filter is registered allowing only vertices attached to the end of the selected curve to be picked. When picking you will be reminded that certain items are being filtered out via an information message in the output window. This can be very helpful when trying to pick vertices that are coincident with other vertices, etc.. This is called a *tool defined* filter. User registered filters are called *user defined*.

An example of a command to register a filter follows:

Register [filter] Surface Geometry_Type
{spline|plane|cone|cylinder|sphere|torus|best fit] [and|or] [include|exclude]

Filtering With Registered Filters

After the filters are registered, they can be exercised with:

Filter <entity> <id range>

For example, if a surface filter was registered (i.e register surface related_to volume 1), the command:

Filter surface all

will display the ids of the surfaces that pass this filter. If there are more than one surface filter is registered, the filter command will consider all of them.

If an entity is to be filtered with respect to other registered entity filters, the range identifier "in" can be used to consider those other entity filters. For example, suppose there are node and face filters registered. To filter hex entities with consideration to the face and node filters, the command would be:

Filter hex in face <id range> in node <id range> {*filter hex in face all in node all*}

The effect of this command will be to first parse and filter the node entities; then to parse and filter the face entities with respect to the nodes that pass the node filter(s). Then to parse and filter (if there are hex filters) the hex entities with respect to the faces that pass the previous face filter(s).

Filter Operations

List

The filter(s) registered can be listed by entity or they can all be listed at one time. To list by entity, the command is:

List <entity_type> Filter {i.e *list curve filter*}

To list all the filters, the command is: *list filter all*

Suppose these commands have been entered,

register curve length equal_to 5 and *register curve mesh_scheme bias*

An example of the *list curve filter* command output is shown below:

	filter type			
1.)	curve_length (= 5)	{include}	active	User Defined
2.)	curve_mesh_scheme (bias)	{or, include}	active	User Defined

Suppress/Resume

Each registered filter is at first "active". The user can make this filter inactive by suppressing the filter. To suppress the filter, the command is:

Filter Suppress <entity> Filter <number>

Using the example above, to suppress the curve_mesh_scheme filter, the command is:

filter suppress curve filter 2

To make it active again, the command is: *filter resume curve filter 2*

All the filters can be suppressed/ unsuppressed at one time with the command:

Filter {suppress | resume} All

Move

If there are multiple filters, these filters can be re-arranged. The command is:

Filter Move <entity> Filter <number> to <number>

Using the curve filters example above, to move the curve_mesh_scheme above the curve_length filter, the command is:

filter move curve filter 2 to 1

The resulting *list curve filter* command would now output:

	filter type			
1.)	curve_mesh_scheme (bias)	{include}	active	User Defined
2.)	curve_length (= 5)	{or, include}	active	User Defined

Delete

Once a filter has been registered, it can be deleted with the command:

Filter Delete <entity_type> Filter <number>

Using the curve filter examples again, to delete the first filter (curve_length), the command is:

filter delete curve filter 1

All the filters can be deleted at one time with the command: *filter delete all*

Register Filter Commands

A list of the currenty available register commands follows:

Register [filter] Curve Related_To {body|volume|surface|vertex <id>} [and|or] [include|exclude]

Register [filter] Curve Geometry_Type {arc|segmented|spline|line|point}

Register [filter] Curve Length {greater_than|greater_than_equal|less_than| equal_to|less_than_equal <value>} [and|or] [include|exclude]

Register [filter] Curve Attached_Surfaces Equal_To <number> [and|or] [include|exclude]

Register [filter] Curve Mesh_Scheme {bias|dualbias|equal|featuresize|morph} [and|or] [include|exclude]

Register [filter] Curve Is_Tolerant [and|or] [include|exclude]

Register [filter] Vertex Related_To {curve|surface|volume|body<id>} [and|or] [include|exclude]

Register [filter] Vertex Attached_Curves Equal_To <number> [and|or] [include|exclude]

Register [filter] Surface Area {greater_than|greater_than_equal|less_than| equal_to|less_than_equal <value>} [and|or] [include|exclude]

Register [filter] Surface Attached_Volumes Equal_To <number> [and|or] [include|exclude]

Register [filter] Surface Mesh_Scheme {map|pave|parallelpave|pentagaon|submap|triangle|trimap | trimesh|tripave} [and|or] [include|exclude]

Register [filter] {surface|volume} Smooth_Scheme <type> [and|or] [include|exclude]
***** *possible smooth schemes are: laplacian (free/fixed), equipotential (free/fixed), isoparametric, centroid area pull (free/fixed), replicate, optimize (free/fixed/jacobian/optms) winslow (free/fixed), randomize*

Register [filter] Surface Sizing_Function {constant|linear|curvature|super|test|exodus| inv_int|interval|none} [and|or] [include|exclude]

Register [filter] Surface Geometry_Type {spline|plane|cone|cylinder|sphere|torus|best fit] [and|or] [include|exclude]

Register [filter] Surface {is_linking|is_source|is_target} [and|or] [include|exclude]

Register [filter] Surface Related_To {vertex|curve|volume|body <id>} [and|or] [include|exclude]

**Register [filter] Volume Range {greater_than|greater_than_equal|less_than|equal_to|
less_than_equal <value>} [and|or] [include|exclude]**

**Register [filter] Volume Mesh_Scheme
{dice|map|plaster|project|pyramid|rotate|sweep|translate}
[and|or] [include|exclude]**

**Register [filter] Volume Related_To {vertex|curve|surface|body} [and|or]
[include|exclude]**

**Register [filter] Body Volume {greater_than|greater_than_equal|less_than|
equal_to|less_than_equal <value>} [and|or] [include|exclude]**

**Register [filter] Body Related_To [vertex|curve|surface|volume<id>] [and|or]
[include|exclude]**

Register [filter] Body Is_Sheet [and|or] [include|exclude]

**Register [filter] Body Contained_Volumes Equal_To <number> [and|or]
[include|exclude]**

**Register [filter] {curve|surface|vertex} Merge_Partners
[equal_to|greater_than|less_than <number>]
[and|or] [include|exclude]**

**Register [filter] {vertex|curve|surface|volume|body} Is_Meshed [and|or]
[include|exclude]**

Register [filter] {curve|surface} Is_Periodic [and|or] [include|exclude]

**Register [filter] {vertex|curve|surface|volume|body|hex|face|edge|node}
Color <color> [and|or] [include|exclude]**

Register [filter] {vertex|curve|surface|volume|body} Visibility [and|or] [include|exclude]

**Register [filter] {curve|surface|volume|body} Interval_Number
{equal_to|greater_than|less_than <number>} [and|or] [include|exclude]**

**Register [filter] {curve|surface|volume|body} Interval_Setting {default|soft|hard}
[and|or] [include|exclude]**

Register [filter] {surface|volume} Smooth_Scheme <type> [and|or] [include|exclude]
***** *possible smooth schemes are: laplacian (free/fixed), equipotential (free/fixed), isoparametric*
centroid area pull (free/fixed), replicate, optimize (free/fixed/jacobian/optms)
winslow (free/fixed), randomize

**Register [filter] {body|volume} Contained_Surfaces Equal_To <number> [and|or]
[include|exclude]**

**Register [filter] {body|volume|surface} Contained_Curves Equal_To <number> [and|or]
[include|exclude]**

**Register [filter] {body|volume|surface} Contained_Curves Equal_To <number> [and|or]
[include|exclude]**

**Register [filter] {hex|face} Quality_Range <metric_name>
{low|high|top|bottom <number> | malformed}**

[and|or] [include|exclude]

**** possible metric names are: aspect ratio, aspect ratio gamma, skew, taper, warpage, element area, stretch, maximum/mininum angle, oddy, folding, jacobian, element volume diagonal ratio, dimension, scaled jacobian*

Register [filter] Hex Related_To {body|volume|node|face| edge <id>} [and|or] [include|exclude]

Register [filter] Hex Full_Hex [and|or] [include|exclude]

Register [filter] Hex Node_Hex [and|or] [include|exclude]

Register [filter] Face Related_To {body|volume|surface|hex|edge|node<id>} [and|or] [include|exclude]

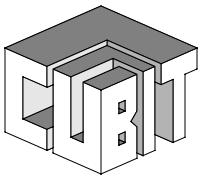
Register [filter] Edge Related_To {body|volume|surface|curve|vertex| hex|face|node<id>} [and|or] [include|exclude]

Register [filter] Node Related_To {body|volume|surface|curve|vertex <id>| hex|face|edge} [and|or] [include|exclude]

Register [filter] Node Position_Fixed [and|or] [include|exclude]

Register [filter] Edge Length {greater_than|greater_than_equal|less_than|equal_to|less_than_equal <value>} [and|or] [include|exclude]

CHAPTER 4: Geometry



Chapter 5: Mesh Generation

▼ Introduction

The methods used to generate a mesh on existing geometry are discussed in this chapter. The definitions used to describe the process are first presented, followed by descriptions of interval specification, mesh scheme selection, and available curve, surface, and volume meshing techniques. The chapter concludes with a description of the mesh editing capabilities, and the quality metrics available for viewing mesh quality, .

Element Types

For each topology type of entity in the model geometry, CUBIT can discretize the entity using one or several types of basic elements; for each order entity in the geometry (vertex, curve, etc.), CUBIT uses a basic element designator to describe the corresponding entity or entities in the mesh. A given geometric topology entity can be discretized with one or several kinds of basic elements in CUBIT. For example, a geometric surface in CUBIT is discretized into a number of faces; this is the basic element designator for surfaces. These faces can consist of two types of basic elements, quadrilaterals or triangles. The basic element designators corresponding to each

type of geometry entity, along with the types of basic elements supported in CUBIT, are summarized in Table 5-1.

Table 5-1: Basic element designators and elements corresponding to geometry entities.

Geometry Entity Type	Basic Element Designator	Basic Element(s) In CUBIT
`Vertex`	Node	Node
`Curve`	Edge	Edge
`Surface`	Face	Quadrilateral, Triangle
`Volume`	Element	Hexahedron, Tetrahedron, Pyramid

For each basic element, CUBIT also supports the definition of several element types, whose use depends on the level of accuracy desired in the target finite element analysis. For example, CUBIT can write both linear (4-noded) and quadratic (8- or 9-noded) quadrilaterals. The element type used for a geometry entity is specified after meshing occurs, as part of the boundary condition specification. See... for a description of that process and the various element types available in CUBIT.

Each mesh entity is associated with a geometry entity which owns it. This associativity allows the user to mesh, display, color, and attach attributes to the mesh through the geometry. For example, setting a mesh attribute on a surface affects all faces owned by that surface.

Mesh Generation Process

Starting with a geometric model, the mesh generation process in CUBIT consists of four primary steps:

1) Set interval size and count for groups of or individual entities

2) Set mesh schemes

3) Generate the mesh for the model

4) Inspect mesh for quality and suitability for targeted analysis

There are also mechanisms for improving mesh quality locally using smoothing and local mesh topology changes. For complex models, this process is usually iterative, repeating steps two to four above.

The mesh for any given geometry is usually generated hierarchically. For example, if the user issues a command to mesh a volume, first its vertices are meshed with nodes, then curves are meshed with edges, then surfaces are meshed with faces, and finally the volume is meshed with hexes. Vertex meshing is of course trivial and thus the user is given little control over this process. However, curve, surface, and volume meshing can be directly controlled by the user.

Each of the steps listed above are now described in detail.

Interval Assignment

Mesh density is usually controlled by the *intervals*, i.e. the number of mesh edges, specified on curves. Intervals are set either directly by specifying the interval count for a curve, or by specifying a desired size for each interval on a curve. Intervals can be specified for curves individually, or indirectly by specifying intervals for higher order geometry containing those curves. Because of interval constraints imposed by various meshing algorithms in CUBIT, the assignment of intervals to curves is not completely arbitrary. For this reason, a global interval match must be performed prior to meshing one or more surfaces or volumes.

Interval Firmness

Before describing the methods used to set and change intervals, it is important that the user understand the concept of interval firmness. An interval firmness value is assigned to a geometry curve along with an interval count or size; this firmness is one of the following values:

hard: interval count is fixed and is not adjusted by interval size command or by interval matching

soft: current interval count is a goal and may be adjusted up or down slightly by interval matching or changed by other interval size commands

default: default firmness setting, used for detecting whether intervals have been set explicitly by the user or by other tools

Interval firmness is used in several ways in CUBIT. Each curve is assigned an interval firmness along with an interval count or size. Commands and tools which change intervals also affect the interval firmness of the curves. Those same commands and tools which change intervals can only do so if the curves being changed have a lower-precedence interval firmness. The firmness settings are listed above in order of decreasing precedence. For example, some commands are only able to change curves whose interval firmness is **soft** or **default**; curves with **hard** firmness are not changed by these commands.

More examples of interval setting commands and how they are affected by firmness are given in the following sections.

A curve's interval firmness can be set explicitly by the user, either for an individual curve or for all the curves contained in a higher order entity, using the command:

{geom_list} Interval {Default | Soft | Hard}

All curves are initialized with an interval firmness of **default**, and any command that changes intervals (including interval assignment) upgrades the firmness to at least **soft**.

Explicit Specification of Intervals

The density of edges along curves is specified by setting the actual number of intervals or by specifying a desired average interval size. The number of intervals or interval size can be explicitly set curve by curve, or implicitly set by specifying the intervals or interval size on a surface or volume containing that edge. For example, setting the intervals for a volume sets the intervals on all curves in that volume.

The commands to specify the number of intervals at the command line are:

{geom_list} Interval <intervals>

{geom_list} [Interval] Size <interval_size>

The first command above sets interval counts. When setting interval counts for surfaces, volumes, bodies and groups, an intervals firmness of **soft** is assigned to the owned curves. When setting the interval count for a curve, a firmness of **hard** is assigned.

Interval size may be specified as well; the interval count for each owned curve is computed by dividing the curve's arc length by the specified interval size. Interval size commands always assign a firmness of **soft** to the specified entities.

The user can scale the current intervals or size with the following commands. Scaling is done on an entity by entity basis.

{geom_list} Interval Factor <factor>

{geom_list} [Interval] Size Factor <factor>

Vertices are not allowed in the **geom_list** for these commands.

Automatic Specification of Intervals

In addition to specifying intervals explicitly based on a known count or size, CUBIT is also able to compute interval counts automatically based on characteristics of the model geometry. The following automatic interval setting command can be used:

{geom_list} Size Auto [Factor <factor>]

Vertices are not allowed in the **geom_list** for this command. Automatic interval assignment works by accumulating the distribution of arc lengths of all owned curves, and assigning an interval size based on a known relative position within that distribution. The user has the option of specifying a relative factor, between zero and eleven, which specifies a position within that arc length distribution. The meaning of various points in this range is summarized in Table 5-2.

Table 5-2: Relative size factors.

Factor	Interpretation
`0`	Size of zero
`1`	Size of smallest curve in entities specified.
`10`	Size of largest curve in entities specified.
`11`	1.5 times size of largest curve in entities specified.

The default factor is six.

The user may assign the interval size to be the arc length of the smallest curve contained in the specified entity or entities using the following command:

{geom_list} Size Smallest curve

Vertices are not allowed in the **geom_list** for this command. This command assigns a soft interval firmness.

Interval Matching

Each meshing scheme in CUBIT imposes a set of constraints on the intervals assigned to the curves bounding the entity being meshed. For example, meshing any surface with quadrilaterals requires that the surface be bounded by an even number of mesh edges. This constrains the intervals on the bounding curves to sum to an even number. For a collection of connected surfaces and volumes, these interval constraints must be resolved globally to ensure that each surface will be meshable with the assigned scheme. The global solution technique implemented in CUBIT is referred to as interval matching.

When meshing a surface or volume, matching intervals is performed automatically. In some cases, interval matching needs to be invoked manually, for example when meshing a collection of volumes, or a collection of surfaces not in a common volume, or when the user has invoked the **control skew** command (detailed in the Mesh Quality section). Interval matching can also be called to check whether the assigned intervals and schemes are compatible.

The command syntax for manually matching intervals is the following:

Match Intervals {geom_list}

Here the entity list can be any mixed collection of groups, bodies, volumes, surfaces and curves.

The interval matcher assigns intervals as close as possible to the user-specified intervals, while satisfying global interval constraints. The goal is to minimize the relative change in pre-assigned intervals on all entities. Interval matching only changes curves with interval firmness of **soft** or **default**.

Extra constraints can be added by the user to improve mesh quality locally; in particular, curves can be constrained to have the same intervals using the command

{curve_list} Interval {Same|Different}

Specifying that curves have the “same” intervals stores them in a set. More curves may be added to an existing set, and sets merged, by future commands. The current contents of the affected sets are printed after each command. A curve may be removed from a set by specifying that its intervals are “different.”

The user may also set interval constraints for groups of curves in relation to other groups of curves using the command

Curve {curve_list} Interval {Equal_to|Greater_than_equal|Less_than_equal} [Curve {curve_list}] [<extra_intervals>]

Thus the set of curves specified first will have either the same, the same or greater, or the same or lesser number of intervals as the second set of curves, if any. If extra intervals are specified, then those intervals will be added to the right-hand-side of the equation.

The interval assignment algorithm tries to find one good interval solution from among the possibly infinite set of solutions. However, if many curves are hard-set or already meshed, there may be no solution. To improve the chances of finding a solution, it is suggested that curves are soft-set whenever possible. Also, a solution might not exist due to the way the local selections of corners and sides of mapped surfaces interact globally. If there is no solution, the following command may help in determining the cause:

Match Intervals {geom_list} [Seed Curve <range>] [Assign Groups [Only|Infeasible]] [Map|Pave]

Specifying **Assign Groups** will create groups that contain independent subproblems of the global problem. Specifying **Assign Groups Only** will group independent subproblems, but the algorithm will not attempt to solve these subproblems. **Assign Groups Infeasible** will put each independent subproblem with no solution into specially named groups. Often poor corner choices and surface meshing schemes will be illuminated this way. If **Map** or **Pave** is specified, then only subproblems involving mapping or paving constraints will be considered. If a **Seed Curve** is specified, then only those subproblems containing that curve will be considered.

Advanced users may also wish to experiment with setting the following, which may change the interval solution slightly:

Set Match Intervals Rounding {on|off}

Set Match Intervals Fast {on|off}

The user can also constrain the parity of intervals on curves:

{geom_list} Interval {Even | Odd}

If **Even** is specified, then during subsequent interval setting commands and during interval assignment, curves are **forced** to have an even number of intervals. If the current number of intervals is odd, then it is increased by one to be even. If **Odd** is specified then intervals **may be either** even or odd. Unless user specified, curves are odd. Setting intervals to even is useful in problems where adjoining faces are paved one by one without global interval assignment.

Periodic Intervals

The number of intervals on a periodic surface, such as a cylinder, in the dimension that is not represented by a curve is usually set implicitly by the surface size. However, periodic intervals and firmness can be specified explicitly by the following commands:

{geom_list} Periodic Interval <intervals>

{geom_list} Periodic Interval {Default|Soft|Hard}

Relative Intervals

If the user needs fine control over mesh density, then for curvy or slanted sides of swept geometries, it is often useful to treat curves as if they had a different length when setting interval sizes. For example, the user may wish to specify that a slanting side curve and a straight side curve have the same “relative length, despite their true length; see Figure 5-1. These are not

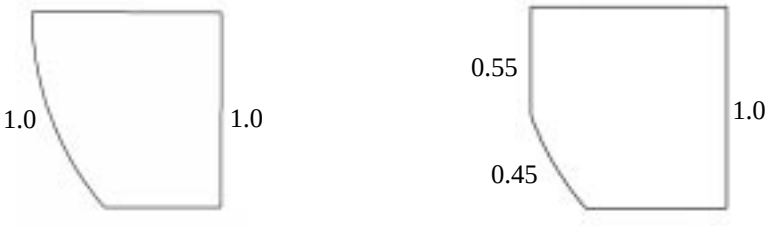


Figure 5-1: Useful relative lengths.

interval matching constraints; interval matching may change intervals so that the user-specified ratio does not hold exactly. The relative lengths of curves are set with the following command:

{geom_list} Relative Length <size>

The following command is used to assign intervals proportional to these lengths:

{geom_list} Relative Interval <base_interval>

For a curve with relative length *x*, setting a relative interval of *y* produces *xy* intervals, rounded to the nearest integer.

Meshing Schemes

Information on specific mesh schemes available in CUBIT is given in this section; the following sections have important meshing-related information as well, and should be read before applying any of the mesh schemes described below.

In most cases, meshing a geometry entity in CUBIT consists of three steps:

1) Set the interval number or size for the entity (see "Interval Assignment" on page 117.)

2) Set the scheme for the object, along with any scheme-specific information, using the scheme setting commands described below.

3) Mesh the object, using the command:

Mesh {geom_list}

This command will match intervals on the given entity (see "Interval Assignment" on page 117), then mesh any unmeshed lower order entities, then mesh the given entity. After meshing is completed, the mesh quality is automatically checked (see "Mesh Quality Assessment" on page 169), then the mesh is drawn in the graphics window.

Bias, Dualbias

Applies to: Curves

Summary: Meshes a curve with node spacing biased toward one or both curve ends.

Syntax:

{curve_list} Scheme Bias {Factor|First_Delta|Fraction} <double>

[Start Vertex <id>]

{curve_list} Scheme Dualbias {Factor|First_Delta|Fraction} <double>

{curve_list} Scheme Bias Fine Size <double>

{Coarse Size <double>|Factor <double>}

[Start Vertex <id>]

{curve_list} Scheme Dualbias Fine Size <double>

{Coarse Size <double>|Factor <double>}

Related Commands:

{curve_list} Scheme Curvature

{curve_list} Scheme Stretch

{curve_list} Reverse Bias

Propagate Curve Biasing [surface|volume|body|group <id_list>]

Discussion:

The Bias scheme allows a mesh that is a different density at one vertex of the curve than the other. The Dualbias scheme allows a mesh that is a different density in the middle of the curve than at the ends. The lengths of the edges follow a geometric series: the length of an edge is equal to a constant factor times the length of the preceding edge, except at the middle of a dualbiased curve. If the factor is greater than one, then, as one moves away from the start vertex, successive edges get longer.

There are four basic interdependent parameters: the number of intervals, the factor, the starting size, and the ending size. The user may specify several combinations of these parameters described below. For scheme Bias, the **start vertex** may also be specified. If no start vertex is specified, the curve's intrinsic start vertex is used as the start vertex; Hint, list the curve.

1) **{curve_list} Scheme Dualbias {Factor|First_Delta|Fraction} <double>**

This syntax assumes that the correct number of edges will be specified by a **size** or **interval** command, and is one of the independent parameters; See "Interval Assignment" on page 117. If **factor** is specified, then this is the ratio between successive edges, and the starting and ending size is whatever makes all of the edge lengths add up to the length of the curve. If **first_delta** is specified, then this is the absolute length of the first edge, and the factor and ending size are dependent. If **fraction** is specified, then the length of the first edge is that fraction of the total curve length, and the factor and ending size is dependent. The syntax for **bias** is similar.

2) **{curve_list} Scheme Dualbias Fine Size <double>**

{Coarse Size <double>|Factor <double>}

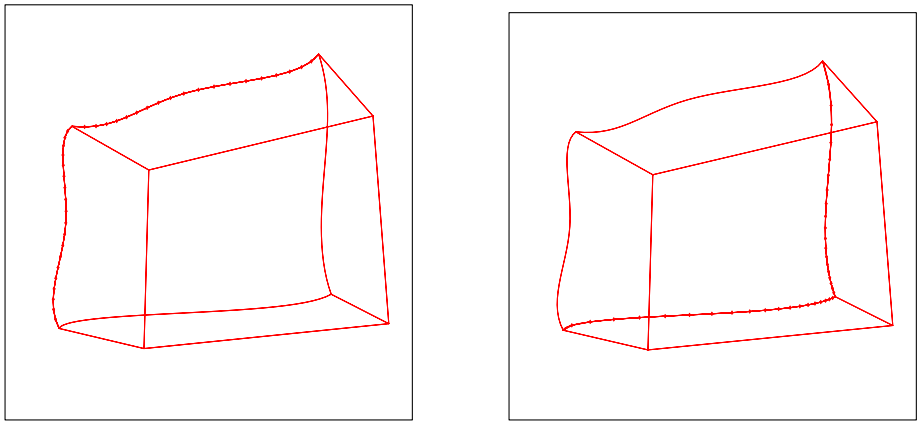
This syntax assumes that the number of **intervals** is a dependent variable: any prior size or interval setting will be overridden by this command, and the number of intervals will be hard set on the curve. The **fine size** is the edge length at the ends, or at the **start vertex** for scheme bias, and is always independent. The **coarse size** is the size at the middle of the curve, or at the non-start vertex; the coarse size may be smaller than the fine size. If the **coarse size** is specified, then the factor is dependent. Otherwise, the **factor** is specified and the coarse size is dependent. The syntax for scheme **bias** is similar.

Scheme stretch is used to keep the number of intervals independent, while specifying *either* or *both* the start and ending size. The edges will not, in general, follow a geometric series.

Scheme curvature is used to keep the number of intervals independent, while adapting the edge lengths to local changes in curvature.

A biased curve can be biased towards the opposite end using the **reversebias** command. The curve may be meshed or unmeshed at this point. Reversing the curve bias using this command is equivalent to setting a bias factor equal to the inverse of the current bias factor.

Figure 5-2 shows the result of meshing two edges with equal and bias schemes.



Equal **Biased**

Figure 5-2: Equal and biased curve meshing

Circle

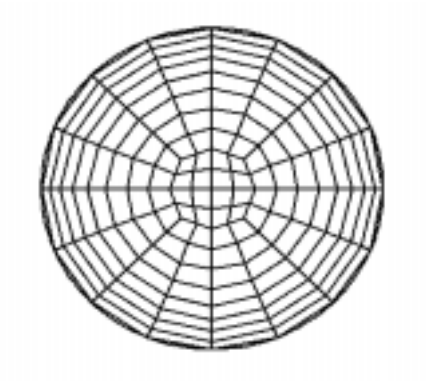


Figure 5-3: Circle Primitive Mesh

Applies to: Surfaces

Summary: Produces a circle-primitive mesh for a surface

Syntax:

{surface_list} Scheme Circle [Interval <int> | Delta_r <double>]

[fraction <double>]

Discussion:

The **Circle** scheme indicates that the region should be meshed as a circle. A “circle”consists of a single bounding curve containing an even number of intervals. Thus, circle can be applied to circles, ellipses, ovals, and regions with “corners” (e.g. polygons). The bounding curve should enclose a convex region. Non-planar bounding loops can also be meshed using the circle primitive provided the surface curvature is not too great. The mesh resembles that obtained via polar coordinates except that the cells at the “center” are quadrilaterals, not triangles. See Figure

5-1 for an example of a circle mesh. Radial grading of the mesh may be achieved via the optional **[intervals]** input parameter or by specifying the radial size **[delta_r]** of the outermost element. **Fraction** has the range 0 < **fraction** < 1 and defaults to 0.5. **Fraction** determines the size of the inner portion of the circle mesh relative to the total radius of the circle.

Copy

Applies to: Curves, Surfaces, Volumes

Summary: Copies the mesh from one entity to another

Syntax:

{curve_list} Scheme Copy source curve <id_range> [[Source Node <starting node id> <ending node id>] [Source Percent [<percentage> | auto]]

[Source [combine|SEPARATE]] [Target [combine|SEPARATE]]

[Source Vertex <id_range>] [Target Vertex <id_range>]]

{surface_list} Scheme Copy [Source Surface <id> Source Curve <id> Target Curve <id> Source Vertex <id> Target Vertex <id> Source Edge <id> Target Edge <id> Source Node <id> Target Node <id>] [Nosmoothing]

{volume_list} Scheme Copy [Source Volume <id> Source Surface <id> Target Surface <id> Source Curve <id> Target Curve <id> Source Vertex <id> Target Vertex <id>] [Nosmoothing]

Related Commands:

Set Morph Smooth {on | off}

Discussion:

If the user desires to copy the mesh from a surface, volume, or a curve or set of curves that has already been meshed, the copy mesh scheme can be used. Note that this scheme can be set before the source entity has been meshed; the source entity will be meshed automatically if necessary before the mesh is copied to the target entity.

The user has the option of providing orientation data to specify how to orient the source mesh on the target entity. For example, when copying a curve mesh, the user can specify which vertex on the source (the source vertex) gets copied to which vertex on the target (the target vertex). If no orientation data is specified, or if the data is insufficient to completely determine the orientation on the target entity, the copy algorithm will attempt to determine the remaining orientation data automatically. If conflicting or inappropriate orientation data is given, the algorithm attempts to discard enough information to arrive at a proper mesh orientation.

Curve mesh copying has certain options that allow the copying of just a section of the source curves' mesh. These options are accessed through the extra keywords given. The percent options allow the user to specify that a certain percentage of the source mesh be copied--in this context the 'auto' keyword means that the percentage will be calculated based on the ratio of lengths of the source and target curves. The combine and separate keywords relate to how the command line is interpreted. If the user wishes to specify a group of target curves that will each receive the copy of the source mesh, then the default **target separate** optionis in effect. If, however, the user wishes the source mesh to be spread out along the range of target curves, then the **target combine** option must be used. The source curves are treated in a similar fashion.

Volume mesh copying depends on the surface copying scheme. Because of this, the target volume must not have any of its surfaces meshed already.

Because of how the copying algorithm works, the target mesh might not be an exact copy of the source mesh. This happens because of the effects of smoothing. If an exact copy is required, there are two possible solutions. The first option is useful when the source and target surfaces or volumes are exact matches. If this criterion is met, the user may specify the Nosmoothing option. That will disable any smoothing of the mesh on the target surface and thereby providing an extremely exact copy of the mesh. The second option is useful if the source and target surfaces are not identical. In this case the user may set the morph smoothing flag on, which will activate a special smoother that will match up the meshes as closely as possible.

Curvature

Applies to: Curves

Summary: Mesh a curve with interval sizes adapted to local curvature.

Syntax:

curve <id_range> scheme curvature <double>

Discussion:

The value of <double> controls the degree of adaption. If zero, the resulting mesh will have nearly equal intervals. If greater than zero, the smallest intervals will correspond to the locations of largest curvature. If less than zero, the largest intervals will correspond to the locations of largest curvature. The default value of **<double>** is zero. Straight lines and circular arcs will produce meshes with near-equal intervals. The method for generating this mesh is iterative and may sometimes not converge. If the method does not converge, either the **<double>** is too large (over-adaption) or the number of intervals is too small. Currently, the scheme does not work on periodic curves.

Dice

Applies to: Curves, Surfaces, Volumes

Summary: Refinement algorithm for refining edges, quads and hexes into smaller ones.

Syntax:

{geom_list} Scheme Dice

Related Commands:

{geom_list} Initialize Dicer

{geom_list} DicerSheet Interval <interval>

DicerSheet <id> interval <interval>

DicerSheet Default Interval <interval>

Replace Mesh {geom_list}

Set Node Constraint {on|off}

Delete Fine Mesh {geom_list} [Propagate]

Discussion:

It is occasionally more convenient to mesh a volume in two stages, first with a coarse mesh and then converting the coarse mesh to a fine mesh. The method used to convert a coarse hex mesh to a fine hex mesh is known as hex dicing.

Hex dicing replaces each coarse element with a grid of smaller elements. The grid is generated by cutting the element any number of times along each of its three primary axes. The number of fine elements in the grid depends on the number of cuts in each direction, and is known as the refinement interval. For example, a hex with a refinement interval of 2 in each direction will be replaced by a grid of 8 smaller elements. A simple example is shown in Figure 5-4.

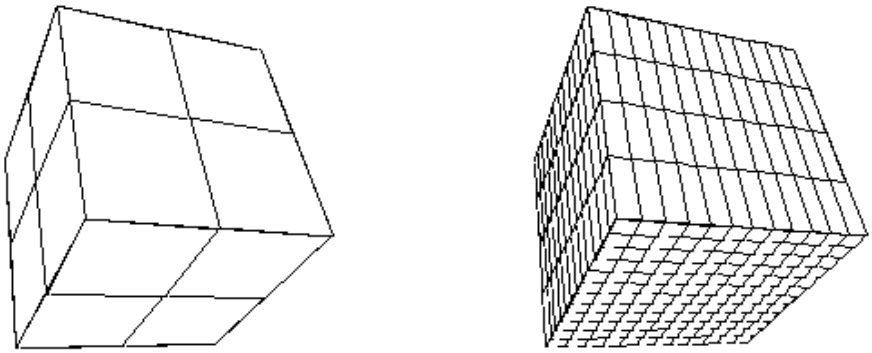


Figure 5-4: Simple Dicing Example

Dicing may also be performed on a quad mesh. The result is a grid of quads replacing each coarse quad element.

In order for the resulting fine mesh to be conformal, groups of coarse mesh edges must have the same refinement interval. Each group of dependent edges is known as a dicer sheet. Dicer sheets often include edges from several surfaces and volumes, so dependencies may propagate throughout the mesh. Dicer sheets are maintained automatically and enforce refinement interval dependencies.

Hex dicing is performed in 4 steps:

1) Initialize the dicer

Before dicing may be carried out, the dicer must first be initialized. This will create the necessary internal data needed to enforce constraints and correctly generate and store the fine mesh. To initialize the dicer for a given entity, use the **Initialize Dicer** command

All appropriate internal data will be generated. If there are dependencies between any of the specified entities, or any entity for which the dicer has already been initialized, those dependencies will automatically be reflected in the internal data with dicer sheets.

2) Set refinement intervals

After the dicer has been initialized, refinement intervals should be set. This will determine the number of fine edges replacing each coarse edge in a given dicer sheet, ultimately determining the number of fine elements that will replace each coarse element. The refinement interval must

be a positive integer, 1 or greater. A refinement interval of 1 will leave the coarse edges essentially unchanged, replacing 1 coarse edge with 1 fine edge.

Refinement intervals may be set on a geometric entity, on individual dicer sheets, or using a default value for all dicer sheets, using the commands:

{geom_list} DicerSheet Interval <interval>

DicerSheet <id> interval <interval>

DicerSheet Default Interval <interval>

The default dicer sheet interval is two.

3) Perform the dicing

Initializing the dicer for an entity will set the mesh scheme for that entity to Dice. Once the scheme has been set to Dice, the coarse mesh can be used to create the fine mesh using the command

Mesh {geom_list}

The fine mesh will be generated and will exist in memory, but at this point will not be applied to the entity that was diced.

4) Replace the coarse mesh with the fine mesh

Once the fine mesh exists in memory, you may replace the coarse mesh with the fine mesh with the command

Replace Mesh {geom_list}

This command works only with surfaces and volumes. Each coarse element will be replaced with its grid of fine elements. As a result, the mesh on any child entities will also be replaced. In other words, replacing the mesh of a volume will also replace the mesh on each of that volume's surfaces and curves.

As a coarse mesh is replaced, any coarse elements that are still needed by another portion of the mesh will not be destroyed. For example, assume that two volumes have been merged and shared a surface. If both volumes are meshed, and the mesh on one volume is then replaced, the shared coarse surface mesh will still exist because it is needed by the other volume. At this point, the surface mesh is in an ambiguous state, simultaneously containing coarse and fine elements. If the second volume is then diced and its mesh is replaced, the coarse mesh on the shared surface will then be deleted and the fine mesh will be conformal between the two volumes.

The Simplified Dicer Commands

The four step process normally used to dice a mesh may be simplified using commands very similar to those used for other meshing schemes. To use the simplified interface, follow these steps:

1) Set the mesh scheme to Dice for each entity to be diced, using a command such as **Volume 1 Scheme Dice**.

2) Set the interval on the entity, using a command such as **Volume 1 Interval 3**. This will set the refinement interval for the specified volumes.

3) Mesh the entity, using a command such as **Mesh Volume 1**.

4) Replace the mesh, using a command such as **Replace Mesh Volume 1**.

While the simplified interface still require four steps, the commands are familiar, similar to those used for other meshing schemes. The simplified commands are automatically converted to the appropriate dicing commands when the situation requires it.

Additional Dicing Commands

Several utilities have been developed to assist the user during the refinement process.

• Constraining Nodes to Geometry

The user can control whether refinement nodes of surface and curve meshes get moved to the geometry, or whether their positions remain as a straight-line interpolation between coarse nodes, via the following command:

Set Node Constraint {on|off}

If **Node Constraint** is on, which is the default, then nodes are constrained to lie on the geometry.

• Deleting a Fine Mesh

The fine nodes generated by the Dicer may be deleted using the command

Delete Fine Mesh {geom_list} [Propagate]

This command only works before using the **Replace Mesh** command. Any fine mesh entities that rely on the deleted fine nodes are also deleted. For example, if the fine nodes on a surface are deleted, the fine mesh of any attached volume is deleted along with the nodes on the surface. If the optional **Propagate** keyword is used, the fine mesh will be deleted from any child entities as well.

• Interaction with Dicer Sheets

Dicer sheets can be drawn, picked, highlighted, and listed, like other entities in the CUBIT model.

Equal

Applies to: Curves

Summary: Meshes a curve with equally-spaced nodes

Syntax:

{curve_list} scheme Equal

Discussion:

See “Interval Assignment” on page 117 for a description of how to set the number of nodes or the node spacing on a curve.

HexToVoid

Applies to: Volumes

Summary: Meshes a volume building hexes from the exterior surfaces of the volume until hexes can no longer be inserted. In general this scheme will not produce a complete (closed) mesh for arbitrary volumes.

Syntax:

volume <volume_id_range> scheme hextovoid

Related Commands:

mesh volume <volume_id_range> [hexes <number_hexes>]

mesh volume <volume_id_range> [layers <number_layers>]

Discussion:

This algorithm is related to plastering, except that the exterior mesh remains fixed. In the general case, this will result in a partial all-hex mesh, and one or more unmeshed interior void regions. The algorithm currently successfully creates a partial mesh for most geometries. HexToVoid is most often used as part of the HexTet meshing algorithm discussed below.

A partial mesh can be created instead of meshing the complete geometry. The number of hex elements or the number of completed layers inward can be specified at the command line.

HexTet

Applies to: Volumes

Summary: Meshes a volume using the **HexToVoid** scheme until hex meshing terminates, finishes remaining void with scheme **TetMesh.**

Syntax:

volume <volume_id_range> scheme hextet

Related Commands:

Set hextet transition_type { two_triangle | four_triangle | pyramid }

Discussion:

This algorithm combines the HexToVoid and TetMesh schemes to create a fully meshed, mixed-element mesh. The HexToVoid algorithm is first invoked to create as many hexes as possible in the volume, working inward from the boundaries. The quadrilateral boundary is then converted to a triangular boundary, and the remaining volume is filled with tetrahedra.

There are three methods currently supported to convert the quadrilateral void boundary to triangles. The two-triangle scheme subdivides each quadrilateral along its shortest diagonal, producing two boundary triangles. The four-triangle scheme inserts a center point on the quadrilateral and divides into four triangles. The pyramid scheme inserts a layer of pyramid elements on top of the quadrilaterals, resulting in a fully-conformal mesh.

Hole

Applies to: Annular Surfaces

Summary: Useful on annular surfaces to produce a “polar coordinates” type mesh (with the singulaity removed).

Syntax:

Surface <surface_id_range> Scheme Hole [rad_intervals <intervals>] [bias <double>] [pair node <loop node-id> with node <loop node-id>]

Discussion:

A polar coordinate-like mesh with the singularity removed is produced with this scheme. The azimuthal coordinate lines will be of constant radius (unlike scheme map) The number of intervals in the azimuthal direction is controlled by setting the number of intervals on the inner and outer bounding loops of the surface (the number of intervals must be the same on each loop). The number of intervals in the radial direction is controlled by the user input, **rad_intervals** (default is one). A bias may be put on the mesh in the radial direction via the input parameter **bias.** A bias of 1 gives a uniform grading (default), a bias less than one gives smaller radial intervals near the inner loop, while a bias greater than one gives smaller radial intervals near the outer loop. The correspondance between mesh nodes on the inner and outer boundaries is controlled with the **pair node <loop node-id> with node <loop node-id>** construct. One id on the inner loop and one id on the outer loop should be given to connect the two nodes by a radial mesh line. If this option is not exercised one risks sub-optimal node pairings, with possible negative Jacobians as the result. To use this option one must first mesh the inner and outer curve looops and determine the mesh node ids.

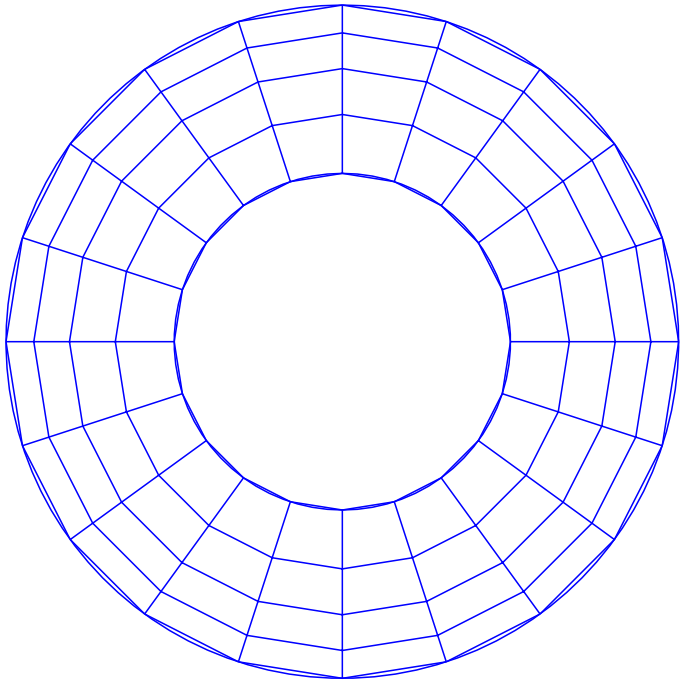


Figure 5-5: Example of Mesh Scheme Hole

Mapping

Applies to: Surfaces, Volumes

Summary: Meshes a surface/volume with a structured mesh of quadrilaterals/hexahedra.

Syntax:

{geom_list} Scheme Map

Discussion:

A structured mesh is defined as one where each interior node on a surface/volume is connected to 4/6 other nodes. Mappable surfaces contain four logical sides and four logical corners of the map; each side can be composed of one or several geometric curves. Similarly, mappable

volumes have six logical sides and eight logical corners; each side can consist of one or several geometric surfaces. For example, in Figure 5-6, the logical corners selected by the algorithm are

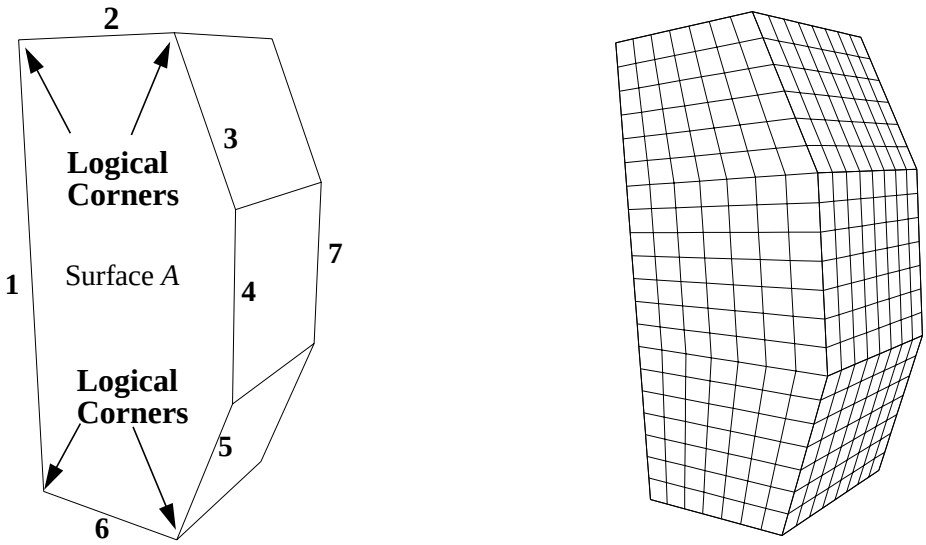


Figure 5-6: Scheme Map Logical Properties

indicated by arrows. Between these vertices the logical sides are defined; these sides are described in Table 5-3. Interval divisions on opposite sides of the logical rectangle are matched

Table 5-3: Listing of logical sides

Logical Side	Curve Groups
Side 1	Curve 1
Side 2	Curve 2
Side 3	Curve 3, curve 4, curve 5
Side 4	Curve 6

to produce the mesh shown in the right portion of Figure 5-6 (i.e. The number of intervals on logical side 1 is equated to the number of intervals on logical side 3). The process is similar for volume mapping except that a logical hexahedron is formed from eight vertices. Note that the corners for both surface and volume mapping can be placed on curves rather than vertices; this allows mapping surfaces and volumes with less than four and eight vertices, respectively. For example, the mapped quarter cylinder shown in Figure 5-7 has only five surfaces.

The choice of where to put the four or eight corners of a mapped mesh is performed automatically, but in some cases produces undesirable results. The user has the option of providing guidance on where to put these corners by specifying surface vertex types; see “Surface Vertex Types” on page 160 for a discussion of this topic. A related discussion on the constraints of the mapping and submapping surface schemes is presented to provide the user with background information about which geometries are most appropriate for these meshing schemes (See “Surface Vertex Types” on page 160).

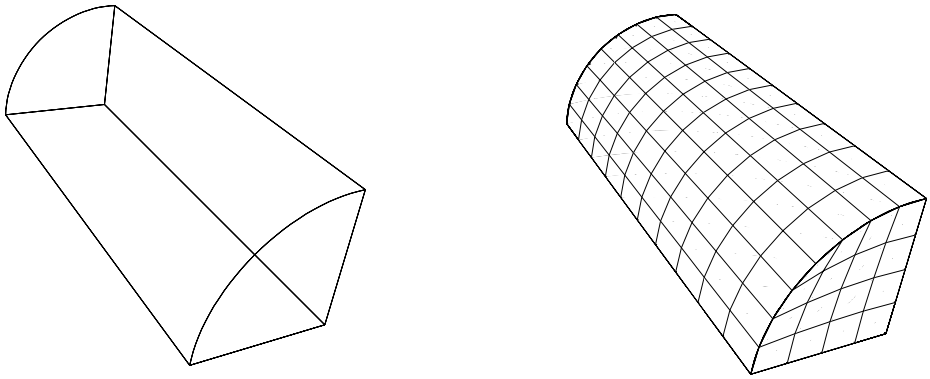


Figure 5-7: Volume mapping of a 5-surfaced volume

In some cases, namely for surfaces or volumes whose boundaries are concave, TFI can produce inverted meshes. In these cases, the mapped meshes can be smoothed to improve quality and yield a non-inverted mesh.

Mirror

Applies to: Surfaces

Summary: Mirrors the mesh from one surface to another

Syntax:

{surface_list} scheme Mirror [Source Surface <id> Source Curve <id> Target Curve <id> Source Vertex <id> Target Vertex <id> Source Edge <id> Target Edge <id> Source Node <id> Target Node <id>] [Nosmoothing]

Discussion:

The **mirror** scheme is very similar to the **copy** scheme and its users should refer to the description of the options under that heading. In order to understand what is changed, a discussion of the copy command is in order. Depending on what the user enters for the copy scheme, the resulting mesh might be oriented one of two ways. For example, if the user entered:

Surface 1 scheme copy source surface 2 source vertex 5 target vertex 1

then the algorithm would match vertex 1 with vertex 5, but then would have to make a guess about how to match the curves. Lacking other pertinent data, the match will be a direct match, as is shown in Figure 5-8.

This default matching can be changed by specifying more information for matching, or the user can specify scheme mirror. The mirror scheme sets up the copying information in such a way as to reverse the default orientation of the target mesh, as is shown in Figure 5-8 (right).

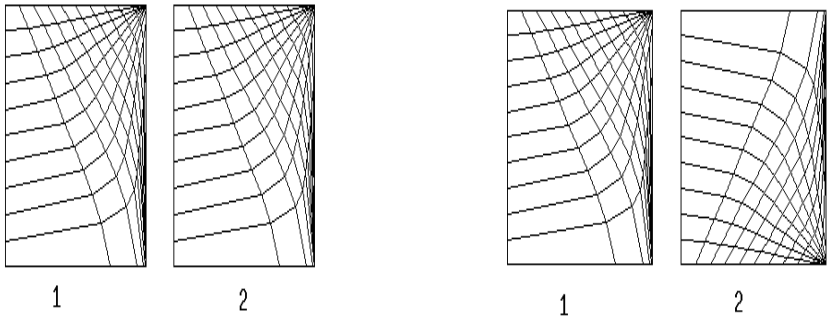


Figure 5-8: Surface 1 copied/mirrored onto surface 2.

Pave

Applies to: Surfaces

Summary: Automatically meshes a surface with an unstructured quadrilateral mesh.

Syntax:

{surface_list} Scheme Pave

Related Commands:

[set] Paver Diagonal Scale <factor (Default = 0.9)>

[set] Paver Grid Cell <factor (Default = 2.5)>

[set] Paver LinearSizing {Off | ON}

[set] Paver Smooth Method {DEFAULT | Smooth Scheme | Old}

Discussion:

Paving (see reference [1]) allows the meshing of an arbitrary three-dimensional surface with quadirlateral elements. The paver supports interior holes, arbitrary boundaries, hard lines, and zero-width cracks. It also allows for easy transitions between dissimilar sizes of elements and element size variations based on sizing functions. Figure 5-9 shows the same surface meshed with mapping (left) and paving (right) schemes using the same discretization of the boundary curves.

When meshing a surface geometry with paving, clean-up and smoothing techniques are automatically applied to the paved mesh. These methods improve the

regularity and quality of the surface mesh. By default the paving method uses its own smoothing methods that are not directly callable from CUBIT. Using one of CUBIT's callable smoothing methods in place of the default method will sometimes improve mesh quality, depending on the surface geometry and specifice mesh characteristics. If the paver produces poor element quality, switching the smoothing scheme may help. This is done by the command:

[set] Paver Smooth Method {DEFAULT | Smooth Scheme | Old}

When the "Smooth Scheme" is selected, the smoothing scheme specified for the surface will be used in place of the paver's smoother. See "Mesh Smoothing" on page 161 for more information about the callable smoothing schemes in CUBIT.

The smoothers flatten elements, such as inserted wedges, that have two edges on the active mesh front. In meshes where this "corner" is a real corner, flatten the element may give an unacceptable mesh. The following command controls how much the diagonal of such an element is able to shrink.

[set] Paver Diagonal Scale <factor (Default = 0.9)>

The range of for the scale factor is 0.5 to 1.0. A scale factor of 1.0 will force the element to be a parallelagram as long as it is on the mesh front. A value of 0.5 will allow the diagonal to be half its calculated lenght. The element may became triangular in shape with the two sides on the mesh front being colinear.

The paver divides the bounding box of a surface into a number of cells based on the average length of an element. It uses these cells to speed intersection checking of new element edges with the existing mesh. If both very long and very short edges fall in the same area, it is possible that a long edge spans the search region as is excluded from the intersection check when it does intersect the new element. The following command allows the user to adjust the size of the grid cells.

[set] Paver Grid Cell <factor (Default = 2.5)>

The grid cell factor is a multiplier applied to the average element size. This gives the grid cell size. The surface's bounding box is divided by this cell size to determine the number of cells in each direction. A larger cell size means each cell contains more nodes and edges. A smaller cell size means each celll has fewer nodes and edges. A larger cell size forces the intersection algorithm to check more potential intersections, which results in long paver times. A smaller cell size gives the intersection algorithm few edges to check (faster execution) but may result in missed intersections where the ratio of long to short element edges is great. Increase this value if the paver is missing intersections of elements.

The paving algorithm will automatically select a "linear" sizing function if the ratio the largest element to the smallest is greater than 6.0 and no other sizing function is specified for the surface. This is usually desireable. When it is not, the user can change this behavior with the command:

[set] Paver LinearSizing {Off | ON}

Setting paver linear sizing to "off" will keep the default behavior. The size of the element will be based on the side(s) of the element on the mesh front. For a discussion of sizing functions see Appendix E.

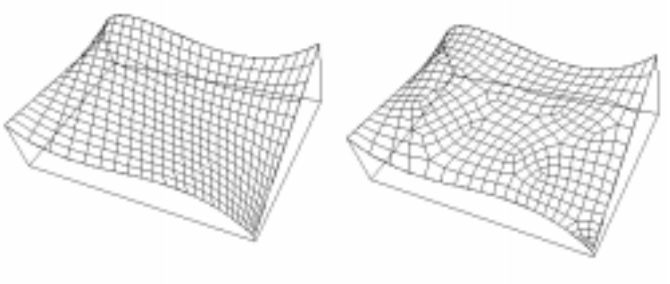


Figure 5-9: Map (left) and Paved (right) Surface Meshes

Pentagon Primitive

Applies to: Surfaces

Summary: Automatically meshes a surface with primitive for 5-sided regions using a block-structured quadrilateral mesh.

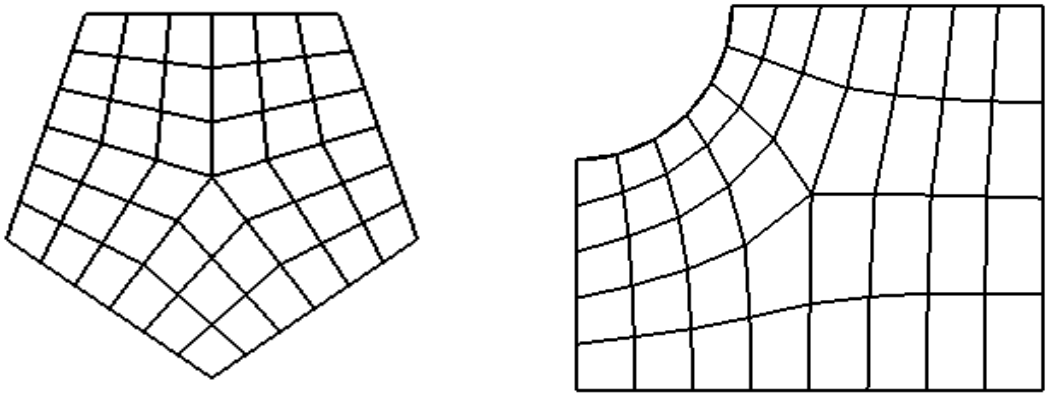
Syntax:

{surface_list} Scheme Pentagon

Related Commands: none

A new meshing primitive for 5-sided regions has been developed. This is similar to the triangle scheme, except it uses 5 sides.

The pentagon scheme indicates the the region should be meshed as a pentagon. The scheme works best if the shape has 5 well-defined corners; however shapes with more corners can be meshed. The algorithm requires that there be at least 10 intervals (2 per side) specified on the curves representing the perimiter of the surface. In addition, the sum of the intervals on any three connected sides must be at least two greater than the sum of the intervals on the remaining two sides. The figure below shows several pentagon meshes.



Plastering

Applies to: Volumes

Summary: Research algorithm for generating all-hexahedral meshes for arbitrary 3D volumes

Syntax:

{volume_list} Scheme Plaster

Related Commands:

Mesh {volume_list} [hexes <number_hexes>] [layers <number_layers>]

Discussion:

Plastering uses the discretized surface and projects elements into the interior of the volume. This continues until the volume fills, with adjustments made to the exterior surface mesh as deemed necessary. This algorithm is currently under development and not suggested for use although it may be tested if desired. It should currently perform well for blocky structures where the surface mesh will form a valid boundary for an interior hex mesh. Some examples of these structures are shown in Figure 5-10. These structures allow very straightforward hex element connectivity

and do not contain any irregular nodes (nodes that are shared by other than four element edges in a given layer).

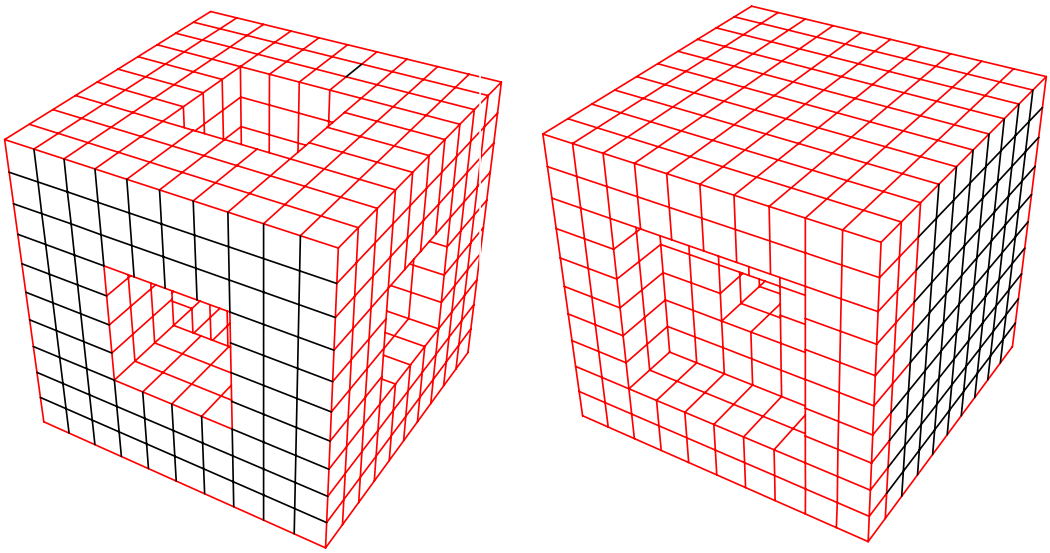


Figure 5-10: Plastering Examples

A partial mesh can be created instead of meshing the complete geometry. The number of hex elements or the number of completed layers inward can be specified at the command line when giving the mesh command by using the following syntax:

mesh {volume_list} [hexes <number_hexes>] [layers <number_layers>]

QTri

Applies to: Surfaces

Summary: Meshes surfaces with the paving algorithm, then converts the quadrilateral elements into triangles.

Syntax:

{surface_list} Scheme QTri

QTri {surface_list}

Related Commands:

Set Node Constraint {On|Off}

Discussion:

QTri is used to mesh surfaces with triangular elements. It uses the quadrilateral paving algorithm first. The quads generated by paving are then split along the diagonal to produce triangles. This command is used as a backup for the TriMesh command, when it fails or can't be used.

The first command listed above sets the meshing scheme on a surface to QTri. The second form sets the scheme and generates the mesh all in one step.

Using QTri on a surface that has already been meshed with quadrilateral elements will split the existing elements into triangles. This feature allows you to use a meshing scheme other than paving to generate the initial quads.

Sphere

Applies to: Volumes topologically equivalent to a sphere and having one surface.

Summary: Generates a radially-graded hex mesh on a spherical volume.

Syntax:

Volume <volume_id_range> Scheme Sphere [graded_interval <int>] [az_interval <int>] [bias <val>] [fraction <val>]

Discussion:

This scheme generates a radially-graded mesh on a spherical volume having a single bounding surface. The mesh is a straightforward generalization of scheme Circle for surfaces. The number of azimuthal intervals around the equator is controlled by the **az_interval** input parameter. The number of radial intervals in the outer portion of the sphere is controlled by the **graded_interval** input parameter. Azimuthal mesh lines in the outer portion of the sphere have constant radius. The inner portion of the volume mesh forms a cube. The **bias** parameter controls the amount of radial grading in the outer portion of the mesh (default=1 gives a uniform

mesh). The **fraction** parameter (between 0 and 1) determines what fraction of the sphere is occupied by the inner cube.

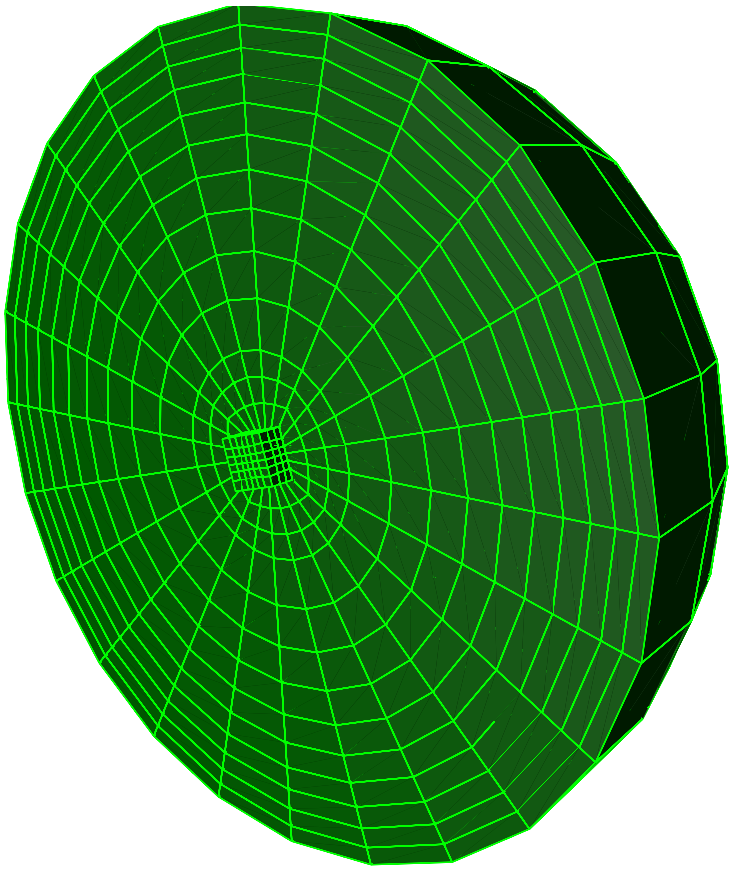


Figure 5-11: Example of Mesh Scheme Sphere

Stretch

Applies to: Curves

Summary: Permits user to specify the exact size of the first and/or last edges on a curve.

Syntax:

Curve <id_range> scheme stretch
[first_size <double> [last_size <double> | stretch_factor <double>]]
[start_vertex <int>]

Related Commands:

{curve_list} Scheme Bias

{curve_list} Scheme Dualbias

Discussion:

This scheme allows the user to specify the exact length of the first and/or last edge on a curve, independent of the number of intervals. Intermediate edge lengths will vary smoothly between these input values, fitting in the required number of intervals. Meshes that are denser or finer in

the middle of the curve than at either end may be created by increasing or decreasing the number of intervals. Unlike scheme bias and dualbias, edges lengths will not, in general, follow a geometric series. Reasonable size parameters should be input: e.g., the sizes must be less than the total length of the curve. If **last_size** or **stretch_factor** is input, **first_size** must also be input. Both **last_size** and **stretch_factor** may not be input. The Stretch scheme currently does not work on periodic curves. The number of intervals on the curve are specified normally with a size or interval command.

Submap

Applies to: Surfaces, Volumes

Summary: Produces a structured mesh for surfaces/volumes with more than 4/6 logical sides

Syntax:

{geom_list} Scheme Submap

Related Commands:

{geom_list} SubMap Smooth <on|off>

Discussion:

Submapping is a meshing tool based on the surface mapping capability discussed previously, and is suited for mesh generation on surfaces which can be decomposed into mappable subsurfaces. This algorithm uses a decomposition method to break the surface into simple mappable regions. Submapping is not limited by the number of logical sides in the geometry or by the number of edges. The submap tool, however is best suited for surfaces and volumes that are fairly blocky or that contain interior angles that are close to multiples of 90 degrees.

An example of a volume and its surfaces meshed with submapping is shown in Figure 5-12.

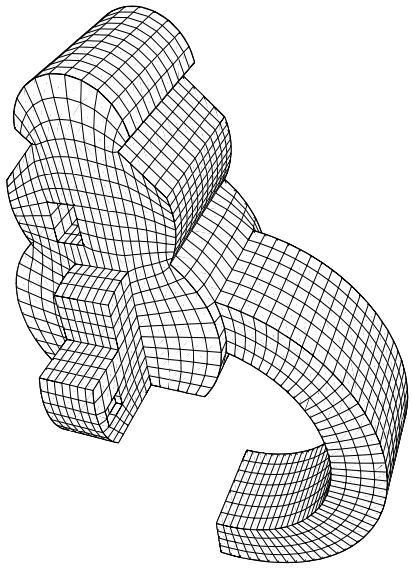


Figure 5-12: Quadrilateral and hexahedral meshes generated by submapping

Like the mapping scheme, submapping uses vertex types to determine where to put the corners of the mapped mesh (see “Surface Vertex Types” on page 160). For surface submapping, curves on the surface are traversed and grouped into “*logical sides*” by a classification of the curves

position in a local “i-j” coordinate system. Volume submapping uses the logical sides for the bounding surfaces and the vertex types to construct a logical “i-j-k” coordinate system, which is used to construct the logical sides of the volume. For surface and volume submapping, the sides are used to formulate the interval constraints for the surface or volume.

Figure 5-13 shows an example of this logical classification technique, where the edges on the

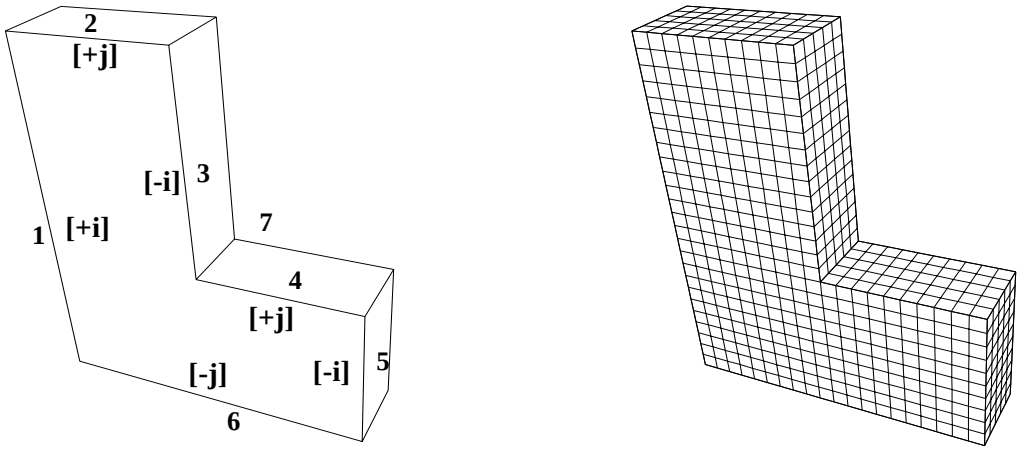


Figure 5-13: Scheme *Submap* Logical Properties

front surface have been classified in the i-j coordinate system; the figure also shows the submapped mesh for that volume.

After submapping has subdivided the surface and applied the mapped meshing technique mentioned above, the mesh is smoothed to improve mesh quality[1]. Sometimes smoothing can decrease the quality of the mesh; in this case the following command can turn off the automatic smoothing before meshing:

{geom_list} SubMap Smooth <on|off>

Surface submapping also has the ability to mesh periodic surfaces such as cylinders; an example of a periodic surface meshed with submapping is shown in Figure 5-14. The requirement for meshing these surfaces is that the top and bottom of the cylinder must have matching intervals. For periodic surfaces, there are no curves connecting the top and bottom of the cylinder; setting intervals in this direction on the surface can be done by setting the periodic intervals for that surface (see “Interval Assignment” on page 117). No special commands need to be given to submap a periodic surface, the algorithm will automatically detect this. Currently, periodic surfaces with interior holes are *not* supported.

Volume submapping is limited to geometries that meet the following two criteria:

1) the bounding surfaces have been meshed with surface submapping or mapping, and

2) three, five, and six valent nodes occur only at junctions where surfaces meet.

1. Because the decomposition performed by submapping is mesh based, no geometry is created in the process and the resulting interior mesh can be smoothed.

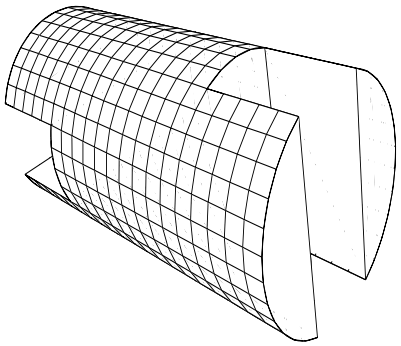


Figure 5-14: Periodic Surface Meshing with Submapping

Sweep

Applies to: Volumes

Summary: Produces an extruded hexahedral mesh for 2.5D volumes.

Syntax:

{volume_list} Scheme Sweep [Source [Surface] <id>] [Target [Surface] <id>]

Related Commands:

{volume_list} Sweep Smooth <on|off|winslow>

Discussion:

The **sweep** algorithm can sweep general 2.5D geometries and can also do pure translation or rotations. Figure 5-15 displays swept meshes involving mapped and paved source surfaces. The

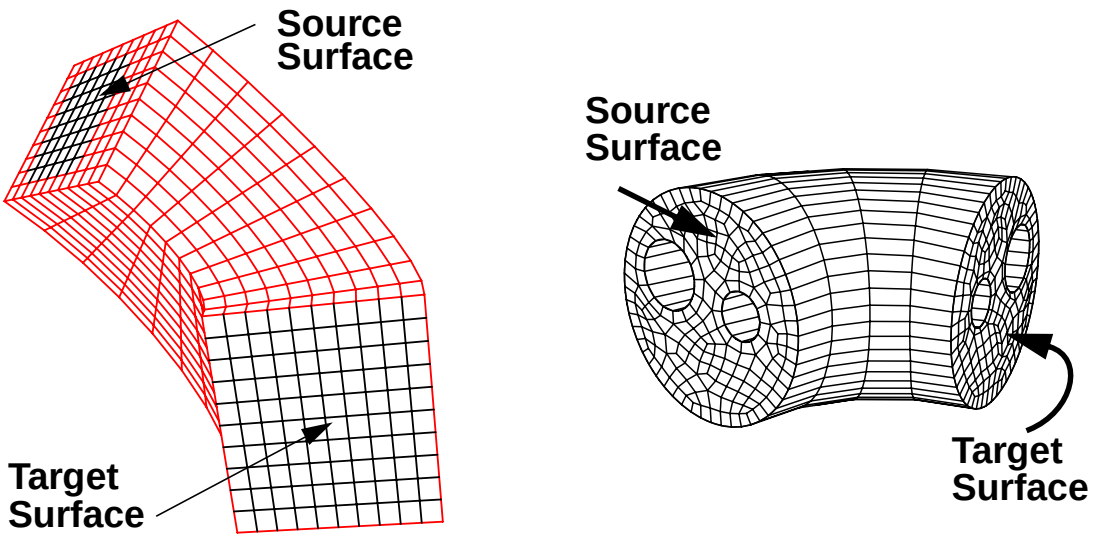


Figure 5-15: Sweep Volume Meshing

sweep algorithm can also handle multiple surfaces linking the source surface and the target surfaces. An example of this is shown in Figure 5-16.

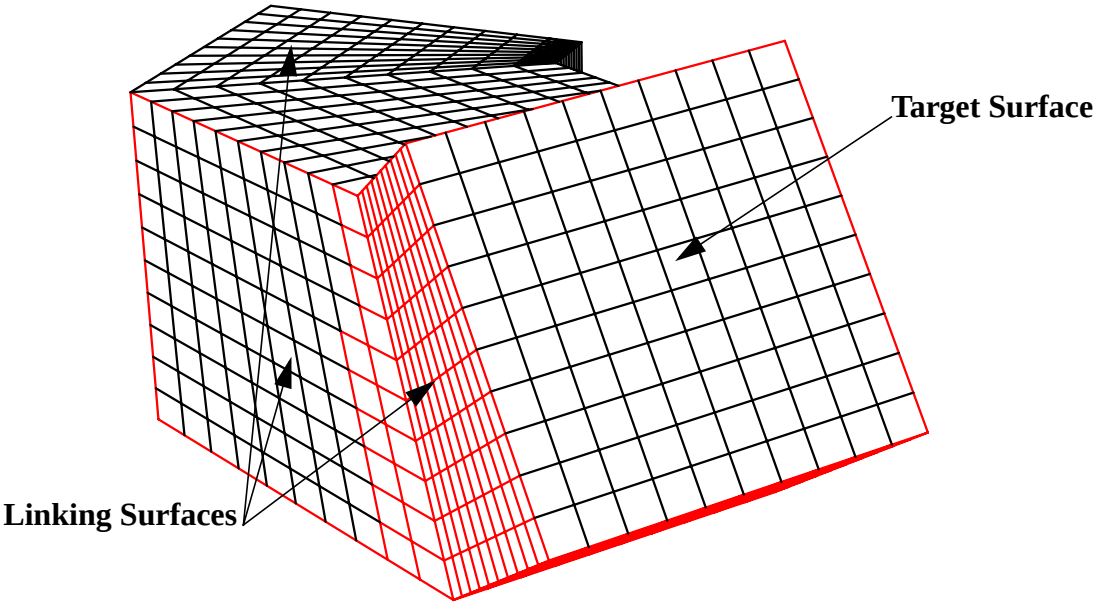


Figure 5-16: Multiple Surface Sweep Volume Meshing

If the source and target surfaces are not specified, then CUBIT attempts to automatically select them. Setting a sweep scheme on a volume automatically selects schemes for the surfaces of the volume. Also, CUBIT automatically sets curve and vertex types in an attempt to make the mesh of the linking surfaces lead from a source surface to a target surface. These automatic selections occasionally fail, in which case the user must manually select the source/target surfaces, some source/target or linking surface meshing schemes, or some curve and vertex types. After making some of these changes, the user should again attempt to set the volume scheme to sweep, etc.

Occasionally the user must also adjust intervals along curves: in addition to the usual surface interval matching requirements, for a given pair of source/target surfaces, there must be the same number of hexahedral layers between them regardless of the path taken. This constrains the number of edges along curves of linking surfaces. For example, in Figure 5-15 right, the number of intervals through the holes must be the same as along the outer shell.

In most cases swept meshes do not require smoothing, however in some cases smoothing is required in each layer of the sweep. In practice, volumes are swept without layer smoothing, and if the resulting mesh quality is poor, the user should turn on layer smoothing. The following command enables layer-smoothing:

{volume_list} Sweep Smooth <on|off|winslow>

The default setting for layer smoothing is *off*. This means that no smoothing will be applied to the layer meshes. If layer smoothing is turned on, a weighted winslow smooth is performed which smooths the layer and preserves biasing of the source mesh in so far as possible. A third option is to set sweep smooth to winslow, which results in unweighted winslow smoothing which does not preserve biasing but sometimes result in better mesh quality.

Some helpful hints in using sweep**:**

1) Sweep runs faster if “sweep smooth” is off. If the geometry/surface mesh permits translation, rotation, or scaling then no smoothing should be needed.

2) The source and linking surfaces of the volume will be automatically meshed if the user has not already meshed them prior to meshing the volume with sweep. *It is important to have high quality meshes on the linking surfaces that are synchronized with one another to that sweep can succeed.* For example, if the geometry suggests translation as the appropriate technique, a translated mesh will still not result from sweep unless the meshes on the volume surfaces are set up accordingly. If there are bad quadrilaterals on the surface meshes, sweep automatically aborts.

3) The target may be meshed by the user or that task may be left to sweep. If the target surface is meshed prior to invoking sweep, then the target mesh must be topologically equivalent to the set of source surface meshes.

4) Biasing of the curve meshes in the direction of the sweep is preserved by the sweep. Biasing of the source mesh boundary is not preserved under a sweep. To accomplish the latter, the user must bias the target surface boundary.

5) The most common error message generated by sweep reads “Target partially reached. Check intervals on Linking Surfaces.” The error-trap that provokes this message is quite general and may occur for a number of reasons, not necessarily the reason given. One of the most frequent causes for this message is a geometry with a thru-hole with the linking surfaces having a different number of intervals on the inside vs. the outside of the volume.

6) If either or both the source and/or target surfaces are omitted from the scheme setting command, CUBIT will determine source and target surfaces (see “Automatic Scheme Selection” on page 157). Sweeping can be further automated using the “sweep groups” command.

7) **Limitations:** Not all geometries are sweepable. Even some that appear sweepable may not be, depending on the linking surface meshes. Highly curved source and target surfaces may not be meshable with the current sweep algorithm. Multiple target surfaces are currently not allowed.

Many-to-Many, or Multisweeping

Applies to: Sweepable volumes with multiple target surfaces

Summary: Extends the sweeping algorithms to include volumes with more than one target surface.

Syntax:

Related Commands:

[set] multisweep smoothing {ON|off}

Discussion:

New to CUBIT version 4.0 is the ability to mesh volumes with multiple target surfaces. The new multisweep algorithm works by recognizing possible mesh and topology conflicts between the source and target surfaces and working to resolve these conflicts through the use of the virtual geometry capabiliies in CUBIT. Figure 5-17 shows some examples of volumes which have been meshed with the multisweep algorithm.

The multisweep algorithm is simply an addition to the regular sweeping algorithms, the multisweep capabilities are accessed by simply specifying scheme sweep and placing multiple target surfaces in the target list. Also, the autoscheme selection algorithm may assign some volumes to be multiswept.

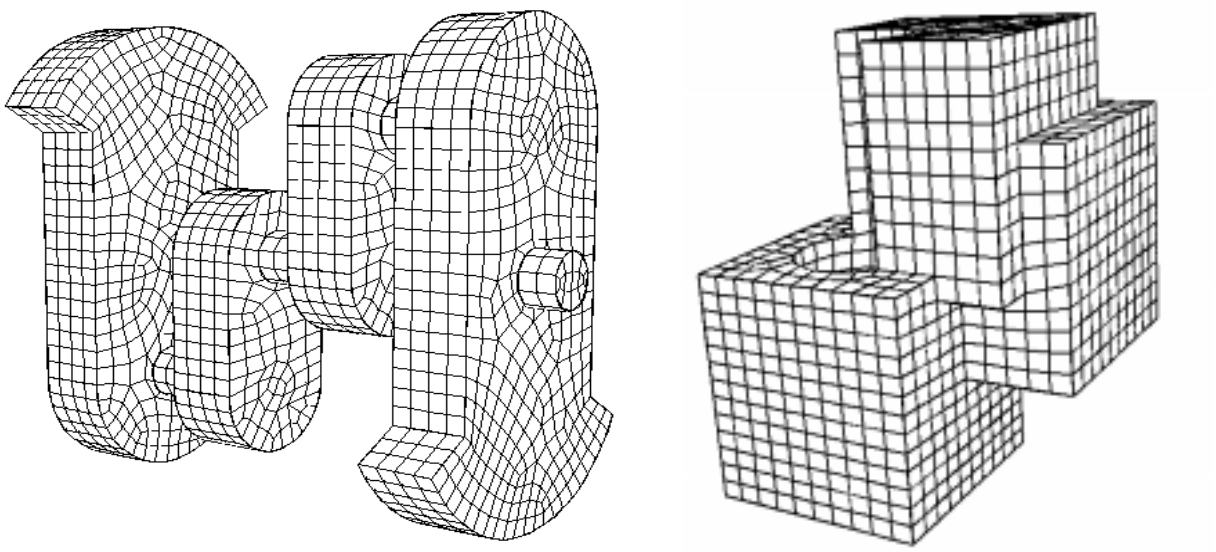


Figure 5-17: Multiswept volume mesh

When setting sizes on volumes which are to be multiswept, it is important to understand that any mesh placed on the volume by a sweeping algorithm is simply a projection of the mesh placed on the source surfaces. It is, therefore, important that similar sizes be placed on both the source and target surfaces to prevent any resulting conflicts when the meshes are aligned during multisweeping and the actual volume meshing.

Because the multisweep algorithm may alter some surface geometry on the volume, it is generally a good idea to attempt to mesh the multisweep volumes first before meshing any other volumes. Also note that this geometry modification may also require some additional scheme selection and interval matching on adjoining volumes.

TetMesh, TetINRIA, TetMSC

Applies to: Volumes

Summary: Automatically meshes a volume with an unstructured tetrahedral mesh.

Syntax:

{volume_list} Scheme {TetMesh | TetINRIA | TetMSC}

Related Commands:

[set] TriMesher {AMG | MSC | INRIA | Simulog}

Discussion:

The "TetMesh" scheme fills and arbitrary three-dimensional volume with tetrahedral elements. The surfaces are first trianglulated with either one

of the triangle schemes (TriMesh, TriAdvance, TriMSC) or a qualrilateral scheme with the qualrilaterals split in two triangles. One or two algorithms are available for generating the tetrahedral mesh.

1. The Simulog/INRIA tetrahedron mesher is included in CUBIT. This is a robust and fast tetrahedron mesher developed in France at INRIA and distributed by Simulog. Figure 5-18 shows a part filled with tetrahedra by this algorithm. You can force this scheme for a volume by giving the command:

{volume_list} Scheme TetINRIA

2. The MSC/AMG Aries tetrahedron mesher may be available. This is an optional component and therefore, not available in all CUBIT installations. You can force this scheme for a volume by giving the command:

{volume_list} Scheme TetMSC

The default tetrahedron meshing scheme is "TetINRIA" since it is included in all versions of CUBIT. The default algorithm may be changed with the command:

[set] TetMesher {AMG | MSC | INRIA | Simulog}

Setting the tetrahedron mesher to "AMG" or "MSC" will select the MSC/AMG Aries tetrahedron mesher as the default algorithm. This is optional software and requires a separate licence, which may not be available. Setting the tetrahedron mesher to "INRIA" or "Simulog" selects that algorithm as the default. All volumes with scheme "TetMesh" will use the Simulog/INRIA algorithm to generate the tetrahedral volume mesh from that point foreward.

Tetrahedron

Applies to: Volumes

Summary: Meshes a four “sided” object with hexahedral elements using the standard tetrahedron primitive.

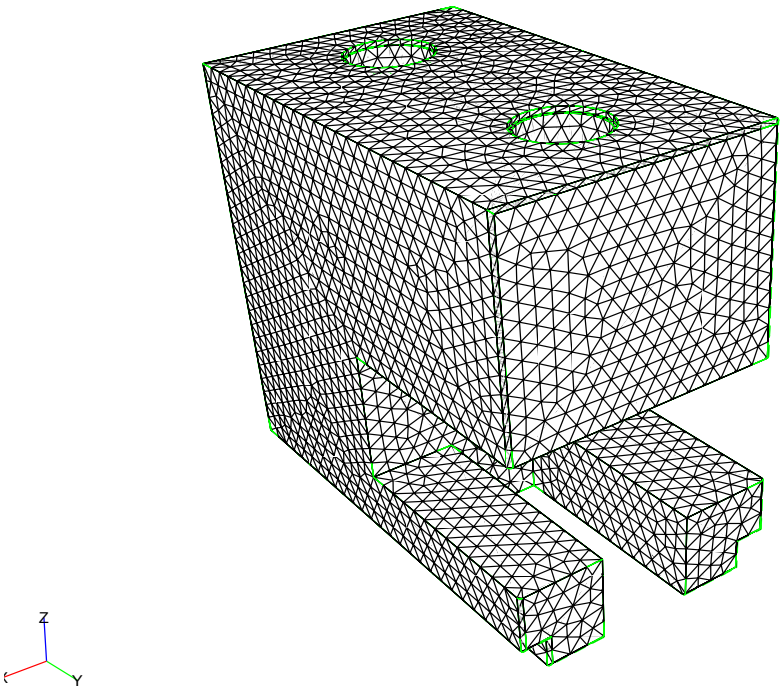


Figure 5-18: Tetrahedral mesh generated with scheme TetMesh

Syntax:

{volume_list} Scheme Tetrahedron [combine surface <range>] [combine surface <range>] [combine surface <range>] [combine surface <range>]

Discussion:

The **tetrahedron** scheme is used to hex mesh volumes with a standard primitive. The primitive assumes that each of the four surfaces have been meshed with the **Triangle** meshing scheme. If more than four surfaces form the tetrahedron geometry, the logical sides can be combined through the **combine** option.

THex

Applies to: Volumes

Summary: Converts tetrahedral elements into hexahedral elements

Syntax:

THex {volume_list}

Related Commands:

Set Node Constraint {On|Off}

Discussion:

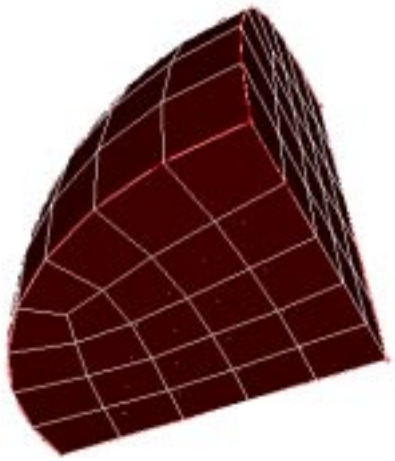


Figure 5-19: Sphere octant hex meshed with scheme Tetrahedron, surfaces meshed using scheme Triangle

The THex command splits each tetrahedral element in a volume into four hexahedral elements, as shown in Figure 5-20. This is done by splitting each edge and face at its midpoint, and then forming connections to the center of the tet.

Note: When THexing merged volumes, all of the volumes must be THexed at the same time, in a single command. Otherwise, meshes on shared surfaces will be invalid.

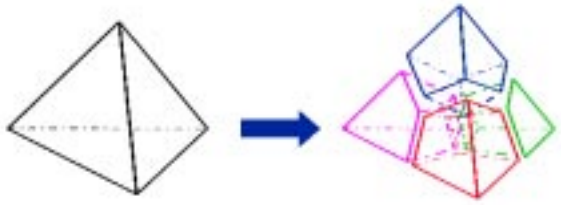


Figure 5-20: Conversion of a tetrahedron to four hexahedra, as performed by the THex algorithm.

An example of the THex algorithm is shown in Figure 5-21.

Transition

Applies to: Surfaces

Summary: Produces a specified transition mesh for specific situations

Syntax:

{surface_list} Scheme Transition {Triangle | Half_circle | Three_to_one | Two_to_one | Convex_corner | Four_to_two} [Source Curve <id>] [Source Vertex <id>]

Discussion:

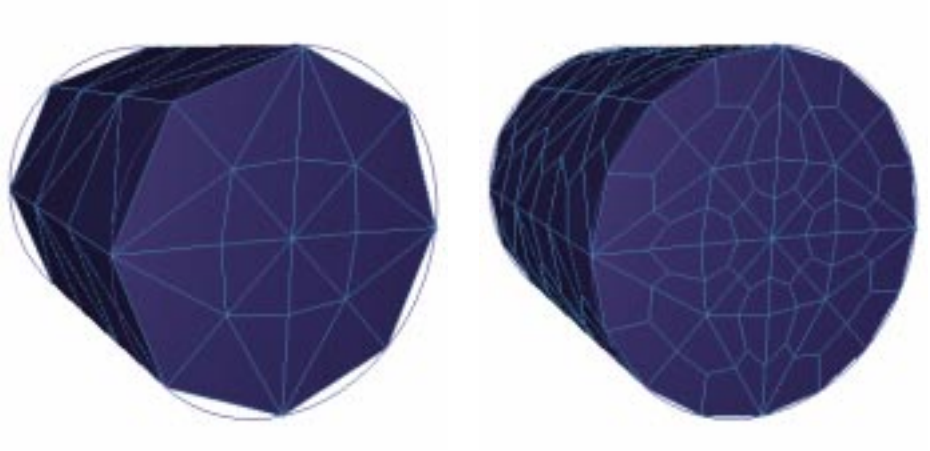


Figure 5-21: A cylinder before and after the THex algorithm is applied.

The **transition** scheme supplies a set of transition primitives which serve to transition a mesh from one density to another across a given surface. The six transition sub-types are demonstrated here.

Scheme Transition Triangle (see Figure 5-22) creates four quads in a triangle that has sides of three, two, and one intervals.

Scheme Transition Half_circle (see Figure 5-22) creates three intervals on the flat and three on the curved part of the half-circle, then creates four quads in the surface.

Scheme Transition Three_to_one (see Figure 5-23) creates four quads on a rectangular surface that has intervals of three, one, one, and one on its sides.

Scheme Transition Two_to_one (see Figure 5-23) creates three quads on a rectangular surface that has intervals of two, two, one and one on its sides

Scheme Transition Convex_corner (see Figure 5-24) takes a six-sided block with a convex corner and connects that inner corner to the opposite one, creating two quads on the surface.

Scheme Transition Four_to_two (see Figure 5-24) creates seven quads on a rectangular surface that has intervals of four, two, two, and two on its sides.

The user also has the option of specifying a source curve and/or a source vertex. The rules for these specifications are as follows:

If both a curve and vertex are specified, the vertex must be on the curve.

The Convex_corner sub-type does not allow a source curve.

The Four_to_two sub-type does not allow a source vertex.

The source curve will be the curve that will be given the fewest intervals.

The source vertex will specify which corner will be used for the scheme, in cases where this makes sense (primarily in the Triangle, and Two_to_one cases).

If none of the optional information is given, the program will assign the source curve to be the shortest one on the face, in keeping with the most probable desires of the user.

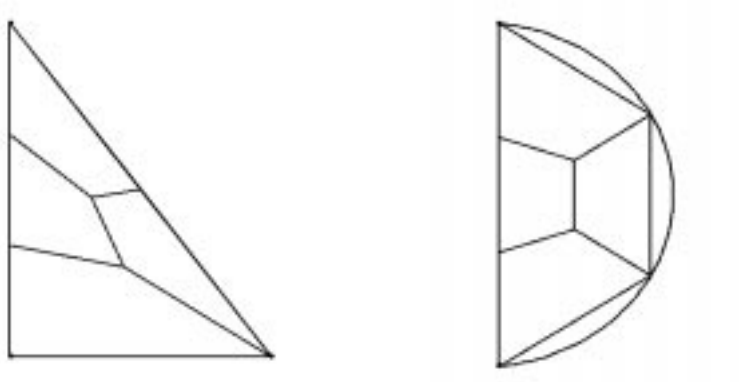


Figure 5-22: Scheme Transition Triangle and Half_circle

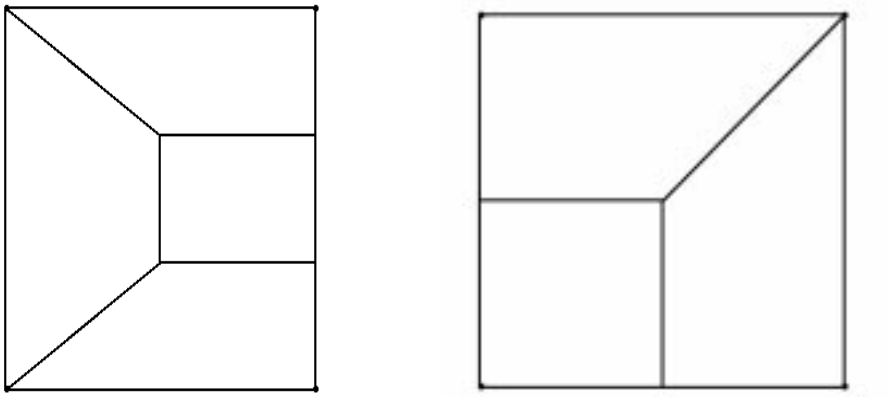


Figure 5-23: Scheme Transition Three_to_one and Two_to_one

Triangle

Applies to: Surfaces

Summary: Produces a triangle-primitive mesh for a surface with three logical sides

Syntax:

{surface_list} Scheme Triangle

Discussion:

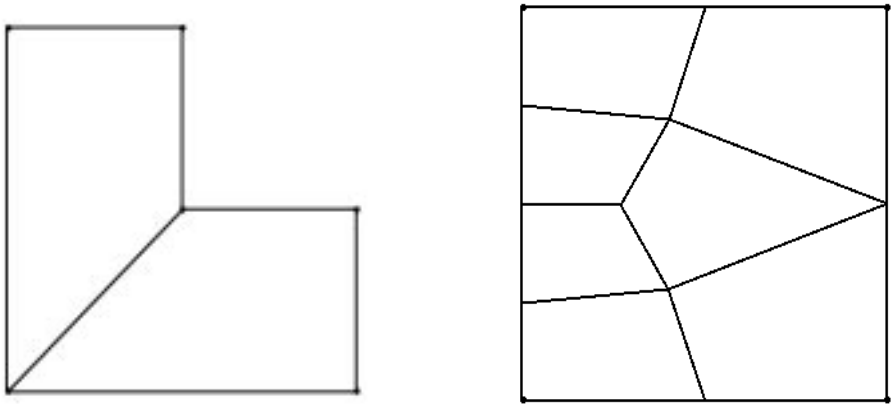


Figure 5-24: Scheme Transition Convex_corner and Four_to_two

The **triangle** scheme indicates that the region should be meshed as a triangle. The definition of the triangle is general in that surfaces containing 3 natural corners can often be meshed successfully with this algorithm. For instance, the surface of a sphere octant is handled nicely by the triangle primitive. The algorithm requires that there be at least 6 intervals (2 per side) specified on the curves representing the perimeter of the surface and that the sum of the intervals on any two of the triangle's sides be at least two greater than the number of intervals on the remaining side. Figure 5-19 on page 149 illustrates a triangle mesh on a 3D surface.

Trimap

Applies to: Surfaces

Summary: Places triangle elements at some vertices, and map meshes the remaining surface.

Syntax:

{surface_list} Scheme Trimap

Related Commands:

{surface_list} {vertex_list} Type {triangle|notriangle}

Discussion:

Some surfaces contain bounding curves which meet at a very acute angle. Meshing these surfaces with an all-quadrilateral mesh will result in a very skewed quad to resolve that angle. In some cases, this is a worse result than simply placing a triangular element to resolve that angle. This scheme resolves this situation by placing a triangular element at such places, and filling the remainder of the surface with a mapped mesh.

The algorithm computes whether a triangular element is necessary, along with where to place that element, automatically. To override the choice of where triangular elements are used, the following command can be used:

{surface_list} {vertex_list} Type {triangle|notriangle}

TriMesh, TriAdvance, TriMSC

Applies to: Surfaces

Summary: Automatically meshes a surface with an unstructured trianglular mesh.

Syntax:

{surface_list} Scheme {TriMesh | TriAdvance | TriMSC}

Related Commands:

[set] TetMesher {AMG | MSC | INRIA | Simulog}

Discussion:

The "TriMesh" scheme fills an arbitrary three-dimensional surface with triangle elements. Three algorithms are available for this purpose.

1. The MSC/AMG Aries tetrahedron mesher has a surface meshing capability. This is an optional component and therefore, not available in all CUBIT installations. You can force this scheme for a surface by giving the command:

{surface_list} Scheme TriMSC

2. An advancing front algorithm is part of the standard CUBIT distribution. It currently allows for holes in the surface and transitions between dissimilar element sizes. It can use a sizing function like the pave scheme if one is defined for the surface. Future development will add hard lines to this scheme’s capabilities. This scheme will be the default "TriMesh" scheme

in the future. You can force this scheme for a surface by giving the command:

{surface_list} Scheme TriAdvance

3. The current default scheme is "QTRI." The "QTRI" scheme first paves the

surface and then cuts the quadrilateral elements in half to form triangles.

Figure 5-25 shows the default "QTRI" mesh (top) and the advancing front mesh (bottom) for the same discretization of the boundary curves.

The default algorithm used by scheme "TriMesh" may be changed with the following command:

[set] TriMesher {AMG | MSC | Advancing Front}

Setting the triangle mesher to "AMG" or "MSC" will select the MSC/AMG Aries tetrahedron mesher as the default algorithm. This is optional software and requires a separate licence, which may not be available. Setting the triangle mesher to "Advancing Front" selects that algorithm as the default. All surfaces with scheme "TriMesh" will use the advancing front algorithm to generate the triangular surface mesh from that point foreward.

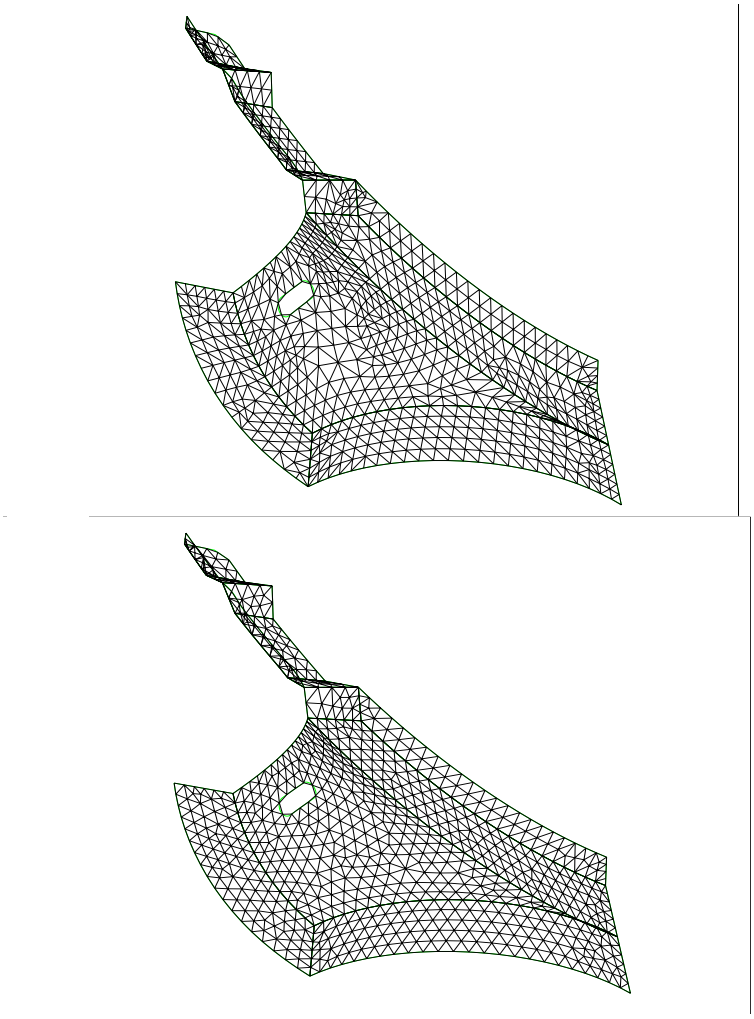


Figure 5-25: Meshes generated with scheme QTRI (top) and TriAdvamce (bottom)

Tripave

Applies to: Surfaces

Summary: Places triangle elements at some vertices, and paves the remaining surface.

Syntax:

{surface_list} Scheme Tripave

Related Commands:

{surface_list} {vertex_list} Type {triangle|notriangle}

Discussion:

Similar to the Trimap algorithm, but uses paving instead of mapping to fill the remainder of the surface with quadrilaterals.

Whisker Weaving

Applies to: Volumes

Summary: Research algorithm for all-hexahedral meshing of arbitrary 3D volumes

Syntax:

{volume_list} Scheme Weave

Related Commands:

Pillow {volume_list}

{geom_list} Mesh [Fixed|Free]

Set AutoWeaveShrink [on|off]

Set Statelist [on|off]

Discussion:

Whisker Weaving is a volume meshing algorithm currently being researched and is not released for general use. However, daring users may find the current form of the algorithm useful for mostly-convex geometries.

Whisker Weaving holds the promise of being able to fill arbitrary geometries with hexahedra that conform to a fixed surface mesh. The algorithm is based on rich information contained in the Spatial Twist Continuum (STC), which is the grouping of the dual of an all-hexahedral mesh into an arrangement of surfaces called *sheets*. Given a bounding quadrilateral surface mesh, Whisker Weaving constructs sheets advancing from the boundary inward. The sheets are then modified so that the arrangement dualizes to a well defined hexahedral mesh. Once the primal hex-mesh is generated, interior node positions are generated by smoothing. Examples of meshes generated using the whisker weaving algorithm are shown in Figure 5-26.

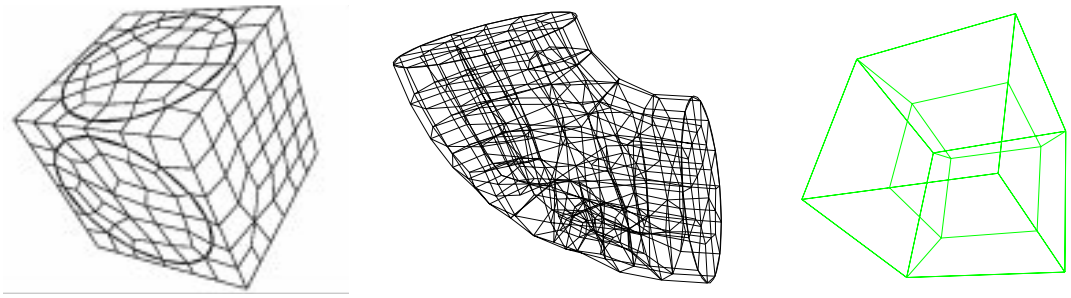


Figure 5-26: Some simple Whisker Weaving meshes with good quality.

Whisker Weaving Basic Commands

The basic steps for meshing a volume with Whisker Weaving are the following:

- Set the meshing scheme for the volume to weave,

  {volume_list} Scheme Weave

- Mesh the volume, which generates hexes,

  Mesh {volume_list}

- **Pillow** the volume to remove certain additional degenerate hexes,

  Pillow {volume_list}

- and typically, smooth the mesh to improve quality, e.g.

{volume_list} Smooth Scheme Optimize Jacobian 1.05

Smooth {volume_list}

Whisker Weaving Options

Currently, Whisker Weaving relies on being able to perturb the bounding quadrilateral mesh. However, a bounding surface's mesh will not be changed if it is contained in another volume that is already meshed. The user may also explicitly prevent Whisker Weaving from changing a bounding mesh by **fixing** it with the following command:

{geom_list} Mesh [Fixed|Free]

The user may select an optional control strategy that doesn't change the surface mesh by setting **AutoWeaveShrink** off, and setting **Statelist** on with the following commands:

Set AutoWeaveShrink [on|off]

Set Statelist [on|off]

Numerous developer commands are available for stepping through the algorithm, examining results, and toggling options. These are available via the online help but are not detailed here.

▼ Automatic Scheme Selection

For volume and surface geometries the user may allow CUBIT to automatically select the meshing scheme. Automatic scheme selection is based on several constraints, some of which are controllable by the user. The algorithms to select meshing schemes will use topological and geometric data to select the best meshing tool. The command to invoke automatic scheme selection is:

{geom_list} Scheme Auto

Specifically for surface meshing, interval specifications will affect the scheme designation. For this reason it is recommended that the user specify intervals before calling automatic scheme selection. If the user later chooses to change the interval assignment, it may be necessary to call scheme selection again. For example, if the user assigns a square surface to have 4 intervals along each curve, scheme selection will choose the surface mapping algorithm. However if the user designates opposite curves to have different intervals, scheme selection will choose paving, since this surface and its assigned intervals will not satisfy the mapping algorithm's interval constraints. In cases where a general interval size for a surface or volume is specified and then changed, scheme selection will not change. For example, if the user specified an interval size of 1.0 a square 10X10 surface, scheme selection will choose mapping. If the user changes the interval size to 2.0, mapping will still be chosen as the meshing scheme from scheme selection. If a mesh density is not specified for a surface, a size based on the smallest curve on the surface will be selected automatically.

Notes: Surface Auto Scheme Selection

Surface scheme selection will choose between Pave, Submap, Triangle, and Map meshing schemes, and will always result in selecting a meshing scheme due to the existence of the paving algorithm, a general surface meshing tool (assuming the surface passes the even interval constraint; see "Interval Matching" on page 119).

Surface auto scheme selection uses an angle metric to determine the vertex type to assign to each vertex on a surface; these vertex types are then analyzed to determine whether the surface can be mapped or submapped (see “Surface Vertex Types” on page 160). Often, a surface’s meshing scheme will be selected as **Pave** or **Triangle** when the user would prefer the surface to be mapped or submapped. The user can overcome this by several methods. First, the user can manually set the surface scheme for the “fuzzy” surface. Second, the user can manually set the “vertex types” for the surface (see “Surface Vertex Types” on page 160). Third, the user can increase the angle tolerance for determining “fuzziness.” The command to change scheme selection’s angle tolerances is:

[Set] Scheme Auto Fuzzy Tolerance {value} (value in degrees)

The acceptable range of values is between 0 and 360 degrees. If the user enters 360 degrees as the fuzzy tolerance, no fuzzy tolerance checks will be calculated, and in general mapping and submapping will be chosen more often. If the user enters 0 degrees, only surfaces that are “blocky” will be selected to be mapped or submapped, and in general paving will be chosen more often.

Notes: Volume Auto Scheme Selection

When automatic scheme selection is called for a volume, surface scheme selection is invoked on the surfaces of the given volume. Mesh density selections should also be specified before automatic volume scheme selection is invoked due to the relationship of surface and volume scheme assignment.

Volume scheme selection chooses between **Map**, **Submap** and **Sweep** meshing schemes. Other schemes can be assigned manually, either before or after the automatic scheme selection.

Volume scheme selection is limited to selecting schemes for 2.5D geometries, with additional tool limitations (e.g. Sweep can currently only sweep from several sources to a single target, *not* multiple targets; see “Sweep” on page 143); this is due to the lack of a completely automatic 3D hexahedral meshing algorithm. If volume scheme selection is unable to select a meshing scheme, the mesh scheme will remain as the default and a warning will be reported to the user.

Volume scheme selection can fail to select a meshing scheme for several reasons. First, the volume may not be mappable and not 2.5D; in this case, further decomposition of the model may be necessary (see “Geometry Decomposition” on page 73). Second, volume scheme selection may fail due to improper surface scheme selection. Volume schemes such as **Map**, **Submap,** and **Sweep** require certain surface meshing schemes, as mentioned previously.

General Notes

In general automatic scheme selection reduces the amount of user input. If the user knows the model consists of 2.5D meshable volumes, three commands to generate a mesh after importing or creating the model are needed. They are:

```
volume all size <value>
volume all scheme auto
mesh volume all
```

The non-trivial, academic model shown in Figure 5-27 was meshed using these three commands

Scheme Firmness

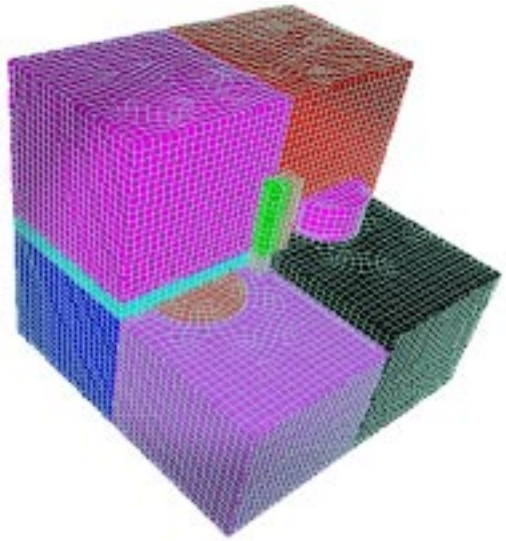


Figure 5-27: Non-trivial model meshed using automatic scheme selection (part of the model is not shown in order to reveal the internal structure of the model)**.**

Meshing schemes may be selected through three different approaches. They are: default settings, automatic scheme selection, and user specification. These methods also affect the scheme firmness settings for surfaces and volumes. Scheme firmness is completely analogous to interval firmness (see “Interval Firmness” on page 117).

Scheme firmness can be set explicitly by the user using the command

{geom_list} Scheme {Default | Soft | Hard}

Scheme firmness settings can only be applied to surfaces and volumes.

This may be useful if the user is working on several different areas in the model. Once she/he is satisfied with an area’s scheme selection and doesn’t want it to change, the firmness command can be given to hard set the schemes in that area. Or, if some surfaces were hard set by the user, and the user now wants to set them through automatic scheme selection then she/he may change the surface’s scheme firmness to **soft** or **default**.

▼ Mesh-Related Topics

There are several topics not related to any one specific mesh scheme, but which are important to understand before using CUBIT to produce meshes. These topics are described in this section.

Grouping Sweepable Volumes

Swept meshing relies on the constraint that the source and target meshes are topologically identical or the target surface is unmeshed (see “Sweep” on page 143.) This results in there being dependencies between swept volumes connected through non-manifold surfaces; these dependencies must be satisfied before the group of volumes can be meshed successfully. For example, if the model was a series of connected cylinders, the proper way to mesh the model would be to sweep each volume starting at the top (or bottom) and continuing through each successive connected volume.

With larger models and with models that contain volumes that require many source surfaces, the process of determining the correct sweeping ordering becomes tedious. The sweep grouping capability computes these dependencies and puts the volumes into groups, in an order which represents those dependencies. The volumes are meshed in the correct order when the resulting group is meshed.

To compute the sweep dependencies, use the command:

Group Sweep Volumes

This will create a group named “sweep_groups”, which can then be meshed using the command:

Mesh sweep_groups

FullHex versus NodeHex Representation

CUBIT has two different internal representations of hexes: FullHexes and NodeHexes. The NodeHex is a lighter weight datastructure, but occasionally nodeset and sideset shortcomings can be overcome by using FullHexes. The user can select which type of hexes get created when generating or importing a volume mesh with the following command:

Set FullHex Use {on|off}

Using the FullHex representation increases the memory used to store a mesh by a factor of approximately five.

Surface Vertex Types

Several meshing algorithms in CUBIT “classify” the vertices of a surface or volume to produce a high quality mesh. This classification is based on the angle between the edges meeting at the vertex, and helps determine where to place the corners of the map, submap or triangle, or the triangles in the trimap or tripave. For example, a surface mapping algorithm must identify the four vertices of the surface that best represent the surface as a rectangle. Figure 5-28 illustrates

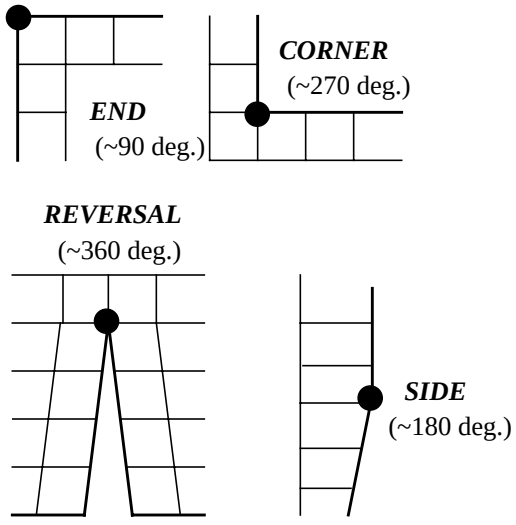


Figure 5-28: Angle Types for Mapped and Submapped Surfaces: An End vertex is contained in one element, a Side vertex two, a Corner three, and a Reversal four.

the vertex angle types for mapped and submapped surfaces, and the correspondence between vertex types and the placement of corners in a mapped or submapped mesh.

The surface vertex type is computed automatically during meshing, but can also be specified manually. In some cases, choosing vertex types manually results in a better quality mesh or a mesh that is preferable to the user. Vertex types are set using the following command:

{surface_list} {vertex_list} Type {end|side|corner|reversal}

Note that a vertex may be connected to several surfaces and its classification can be different for each of those surfaces.

The influence of vertex types when mapping or submapping a surface is illustrated in Figure 5-29. There, the same surface is submapped in two different ways by adjusting the vertex types of ten vertices.

The user may specify the maximum allowable angle at a corner with the following command:

Set {Corner|End} Angle <degrees>

The user may also give greater priority to one automatic selection criteria over the others by changing the following absolute weights. The “corner weight” considers how large angles are at corners. The “turn weight” considers how L-shaped the surface is. The “interval weight” considers how much intervals must change. The “large angle weight” affects only auto-scheme selection: surfaces with a large angle will be paved instead. Each weight’s default is 1 and must be between 0 and 10. The bigger a weight the more that criteria is considered.

Set Corner Weight <value>

Set Turn Weight <value>

Set Interval Weight <value>

Set Large Angle Weight <value>

An illustration of a mesh produced by the submapping algorithm is shown in Figure 5-29. The meshes produced by submapping on the left and right result from adjusting the vertex types of the eight vertices shown.

Preview Mesh

It is sometimes useful to view the nodal locations on curves graphically before meshing (which can take considerably more time). The command to do this is:

Preview Mesh {body|volume|surface|curve|vertex} <id_range>

To clear the display of the temporary nodes, simply issue a “display” command.

▼ Mesh Smoothing

After generating the mesh, it is sometimes necessary to modify that mesh, either by changing the positions of the nodes or by removing the mesh altogether. CUBIT contains a variety of mesh smoothing algorithms for this purpose. Node positions can also be fixed, either by specific node or by geometry entity, to restrict the application of smoothing to non-fixed nodes. There

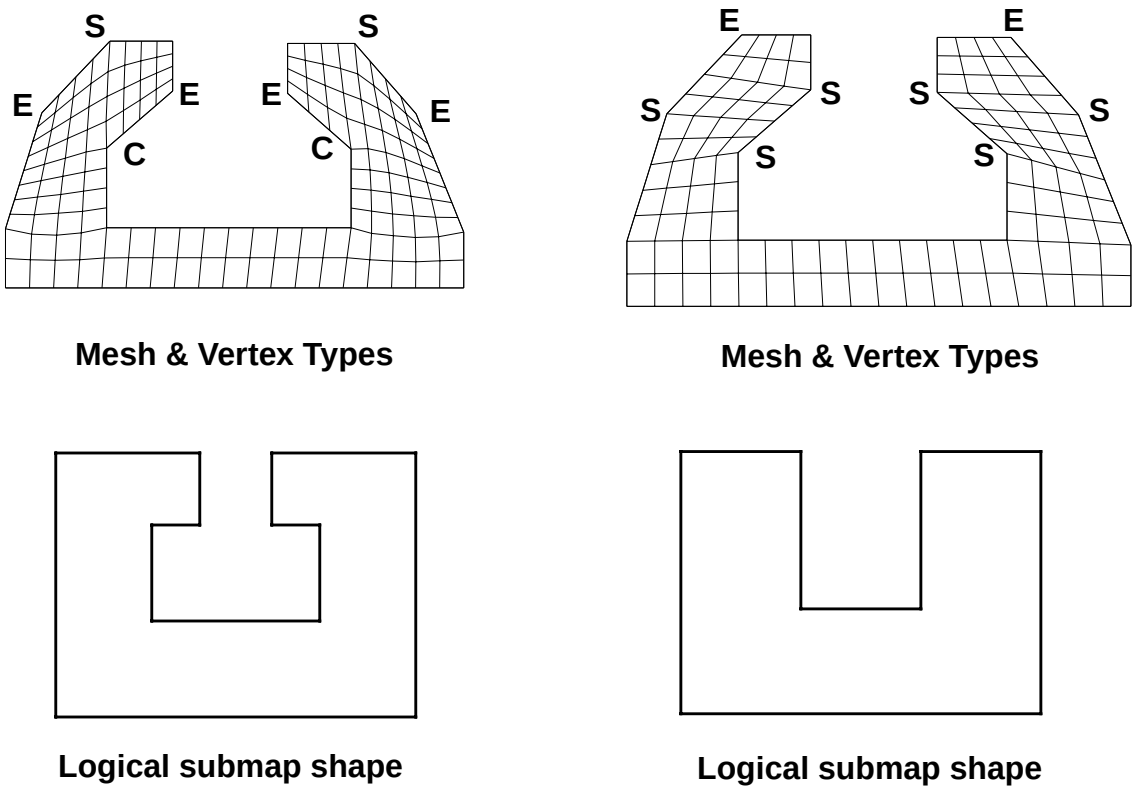


Figure 5-29: Influence of vertex types on submap meshes; vertices whose types are changed are indicated above, along with the mesh produced; logical submap shape shown below.

are also some procedures that can be done before meshing to increase the quality of the mesh. See the section on Mesh Quality for more information.

Mesh smoothing in CUBIT operates in a similar fashion to mesh generation, i.e. it is a two-step process whereby a smooth scheme is chosen and set, then a smooth command performs the actual smoothing. Like meshing algorithms, there is a variety of smoothing algorithms available, some of which apply to multiple geometry entity types and some which only apply to one specific type (these algorithms are described below.) To smooth the mesh on a geometry entity, the user must perform the following steps:

1) Set the smooth scheme for the object, along with any scheme-specific information, using the smooth scheme setting commands described below.

2) Smooth the object, using the command:

Smooth {geom_list}

Groups of entities may be smoothed, by smoothing a group or a body.

If a Body is specified, the volumes in that Body are smoothed. If a Group is specified, only the volume meshes within these groups aresmoothed - no smoothing of the suface meshes is performed.

Typically, smoothing algorithms move nodes in order to improve the quality of the mesh on a given geometry entity. Smoothing is terminated either by satisfying a smoothing tolerance or by

performing the maximum number of smoothing iterations. The smooth tolerance may be set by the user:

[Set] Smooth Tolerance <tol>

The value **<tol>** tells the smoother to stop when node movement is less than tol * the local minimum edge length. The default value for tol is 0.05. The maximum number of iterations may be set by the user: For volumes, the smooth tolerance and iterations may be set by the user but they are presently ignored by the smoothers:

Volume Smooth Tolerance <tol>

Volume Smooth Iterations <iters>

The default values are 0.05 for the tolerance and 18 * (number of hexes / number of nodes)^(1/3)

Where used in the smooth schemes below, the **Free** keyword permits the nodes lying on the bounding entities to "float" along those entities; without this keyword, boundary nodes remain fixed.

Nodal positions may be **fixed** so that no smoothing scheme, either implicit or explicit, will move them, with the following command:

{geom_list} Node Position [Fixed|Free]

The following command does not fix nodal positions, but does fix the connectivity of the mesh, preventing certain volume schemes from changing the bounding mesh:

{geom_list} Mesh [Fixed|Free]

The specific smooth schemes available in CUBIT are now described in detail.

Smooth Scheme: Centroid Area Pull

Applies to: Surface Meshes

Summary: Attempts to create elements of equal area

Syntax:

{surface_list} Smooth Scheme Centroid Area Pull [Free] [Global]

Discussion:

This smooth scheme attempts to create elements of equal area. Each node is pulled toward the centroids of adjacent elements by forces proportional to the respective element areas [8].

Using the **global** keyword when smoothing a group of surfaces will allow smoothing of mesh on shared curves to improve the quality of elements on both surfaces sharing that curve.

Smooth Scheme: Equipotential

Applies to: Volume Meshes

Summary: Attempts to equalize the volume of elements attached to each node

Syntax:

{volume_list} Smooth Scheme Equipotential [Free]

Discussion:

This smoother is a variation of the **Equipotential** [8] algorithm that has been extended to manage non-regular grids [9]. This method tends to equalize element volumes as it adjusts nodal locations. The advantage of the equipotential method is its tendency to “pull in” badly shaped meshes. This capability is not without cost: the equipotential method may take longer to converge or may be divergent. To impose an equipotential smooth on a volume, each element must be smoothed in every iteration—a typically expensive computation. While a Laplacian method can complete smoothing operations with only local nodal calculations, the equipotential method requires complete domain information to operate.

Smooth Scheme: Laplacian

Applies to: Curve, Surface, and Volume meshes

Summary: Tries to make equal edge lengths

Syntax:

{geom_list} Smooth Scheme Laplacian [Free] [Global]

Discussion:

The length-weighted laplacian smoothing approach calculates an average element edge length around the mesh node being smoothed to weight the magnitude of the allowed node movement [8]. Therefore this smoother is highly sensitive to element edge lengths and tends to average these lengths to form better shaped elements. However, similar to the mapping transformations, the length-weighted Laplacian formulation has difficulty with highly concave regions.

Currently, the stopping criterion for curve smoothing is 0.005, i.e., nodes are no longer moved when smoothing moves the node less than 0.005 * the minimum edge length. The maximum number of smoothing iterations is the maximum of 100 and the number of nodes in the curve mesh. Neither of these parameters can currently be set by the user.

Using the **global** keyword when smoothing a group of surfaces will allow smoothing of mesh on shared curves to improve the quality of elements on both surfaces sharing that curve.

Smooth Scheme: Optimize Area

Applies to: Surface meshes

Summary: Produces smooth variation of element across a mesh

Syntax:

{surface_list} Smooth Scheme Optimize Area

Discussion:

Scheme **Optimize Area** generates a mesh by minimizing the sum of the squares of the area of the local elements attached to a node. These meshes often have positive jacobians and give good gradations of element area across a mesh, but are not smooth in an elliptic sense.

Smooth Scheme: Optimize Condition Number

Applies to: Surface and Volume meshes

Summary: Optimizes the mesh condition number to produce well-shaped elements.

Syntax:

Surface <surface_id_range> Smooth Scheme Optimize Condition Number Fixed
Volume <volume_id_range> Smooth Scheme Optimize Condition Number Fixed

Related Commands:

Optimize Untangle

Discussion:

Condition Number measures the distance of an element from the set of degenerate elements. Optimization of the Condition number produces smooth, well-shaped elements when possible. Condition number optimization requires that the initial mesh contain no negative Jacobians (run smooth scheme optimize untangle as a pre-processor). Optimization of Condition Number then guarantees that the optimized mesh will contain no negative Jacobians. Currently, the scheme is being modified to extend the guarantee to the boundary of the mesh.

Smooth Scheme: Optimize Jacobian

Applies to: Volume meshes

Summary: Produces locally-uniform hex meshes by optimizing element Jacobians

Syntax:

{volume_list} Smooth Scheme Optimize Jacobian [param]

Discussion:

The **Optimize Jacobian** method minimizes the sum of the squares of the Jacobians (i.e., volumes) attached to the smooth node. Meshes smothed by this means tend to have locally-uniform hex volumes.

The parameter **<param>** has a default value of 1, meaning that the method will attempt to make local volumes equal. The parameter, which should always be between 1 and 2 (with 1.05 recommended), can be used to sacrifice local volume equality in favor of moving towards meshes with all-positive Jacobians.

Smooth Scheme: Optimize Untangle

Applies to: Surface and Volume meshes

Summary: Untangle a mesh to remove negative Jacobian elements.

Syntax:

Surface <surface_id_range> Smooth Scheme Optimize Untangle Fixed <double>

Volume <volume_id_range> Smooth Scheme Optimize Untangle Fixed <double>

Related Commands:

Optimize Condition Number

Discussion: Occasionally one of the meshing schemes will produce elements having negative Jacobians. Optimize untangle will automatically move mesh nodes to produce meshes with no

negative Jacobian elements. Nodes connected to positive Jacobian elements are not moved. Although the resulting mesh may have no negative Jacobians, the mesh will not be smooth (to further improve the mesh use **Optimize Condition Number**). The optional input parameter **<double>** tells the untangler to halt when the minimum scaled Jacobian is roughly the value of **<double>**. The default value of the parameter is 0 and the range is -1 to 1. In some cases it is not possible to make the minimum scaled Jacobian greater than <double>, in which case the optimizer does the best it can before exiting gracefully.

Smooth Scheme: Randomize

Applies to: Curve, Surface, and Volume meshes

Summary: Randomizes the placement of nodes on a geometry entity

Syntax:

{geom_list} Smooth Scheme Randomize [percent]

Discussion:

This scheme will create non-smooth meshes. If a percent argument is given, this sets the amount by which nodes will be moved as a percentage of the local edge length. The default value for percent is 0.40. This smooth scheme is primarily a research scheme to help test other smooth schemes.

Smooth Scheme: Winslow

Applies to: Surface meshes

Summary: Elliptic smoothing technique for structured and unstructured surface meshes

Syntax:

{surface_list} Smooth Scheme Winslow [Free]

Discussion:

Winslow elliptic smoothing is based on solving Laplaces equation with the independent and dependent variables interchanged. The method is widely used in conjunction with the mapping and submapping methods to give smooth meshes with positive Jacobians, even on non-convex two-dimensional regions. The method has been extended in CUBIT to work on unstructured meshes.

▼ Mesh Deletion

Meshing a complex model often involves iteration between setting mesh parameters, meshing, and checking mesh quality. This often requires removing mesh, for only an entity or for an entity and all its lower order geometry, or sometimes for the entire model.

The command to remove all existing mesh entities from the model is:

Delete mesh

The command for deleting mesh on a specific entity is:

Delete mesh {geom_list} [Propagate]

These commands automatically cause deletion of mesh on higher dimensional entities owning the target geometry.

If the Propogate keyword is used, mesh on lower order entities is deleted as well, but only if that mesh is not used by another higher order entity. For example, if two surfaces (surfaces 1 and 2) sharing a single curve are meshed, and the command **"delete mesh surface 1 propagate"** is entered, the mesh on surface 1 is deleted, as well as the mesh on all the curves bounding surface 1 except the curve shared by surface 2.

In some cases, the capability to delete individual mesh faces on a surface is needed. Deleting a mesh face involves closing a face by merging two mesh nodes indicated in the input. The syntax for this command is:

Delete Face <face_id> Node1 <node1_id> Node2 <node2_id>

This command is provided primarily for developers' use, but also provides the user fine control over surface meshes. At the present time, this command works only with faces appearing on geometric surfaces and should be used before any hex meshing is performed on any volume containing the face to be deleted.

▼ Node and NodeSet Repositioning

A capability to reposition nodesets and individual nodes is provided. This capability will retain all the current connectivity of the nodes involved, but it cannot guarantee that the new locations of the moved nodes do not form intersections with previously existing mesh or geometry. This capability is provided to allow the user maximum control over the mesh model being constructed, and by giving this control the user can possible create mesh that is self-intersecting. The user should be careful that the nodes being relocated will not form such intersections.

The user can reposition nodes appearing in the same nodeset using the **NodeSet Move** command. Moves can be specified using either a relative displacement or an absolute position. The command to reposition nodes in a nodeset is:

{nodeset_list} move <delta_x> <deleta_y> <deleta_z>

{nodeset_list} move to <x_position> <y_position> <z_position>

The first form of the command specifies a relative movement of the nodes by the specified distances and the second form of the command specifies absolute movement to the specified position.

Individual nodes can be repositioned using the Node Move comand. Moves are specified as relative displacements. The command syntax is:

Node <range> Move <delta_x> <deleta_y> <deleta_z>

▼ Mesh Importing and Duplicating

Meshes in Exodus II format may be imported from a file and associated with geometry. The mesh file may have been created by CUBIT, or from some other tool such as Grepos or Patran.

Meshes may also be copyied from one geometric entity to another.

Importing mesh from an external file

see “Nodeset Associativity Data” on page 182.)

Import Mesh ’<exodusII_filename>’

[Block <block_id> [Volume <volume_id>]] [Preview]

Related Commands:

Delete Mesh Preview

Export [Genesis | Mesh] ’<filename>’

List Import Mesh NodeSet Associativity

List [Export Mesh] NodeSet Associativity

set Import Mesh NodeSet Associativity [On|Off]

set [Export Mesh] NodeSet Associativity [On|Off]

The user can import a mesh from an ExodusII file and associate the mesh with matching geometry. This mesh may then be manipulated normally; for example it may be smoothed or part of it deleted and remeshed. The user can save his work by exporting the geometry and mesh, and the importing the geometry and mesh later. In some cases this is faster or more reliable than replaying journal files. Also, teams working on creating a conforming mesh of a large assembly can pass information to one another: a team member can export the mesh on the surfaces between two parts, then another team member import that mesh.

If an exported CUBIT mesh is going to be imported back onto the exact same geometry, then before exporting the user should **set export mesh nodeset associativity on**. This causes extra nodeset data to be written, which associates every node to a geometric entity, so that importing the mesh is more reliable. See “Nodeset Associativity Data” on page 182. When importing, if the user does not want to use the nodeset associativity data that exists in a file, e.g., because the geometry is no longer identical since curves have been composited, or CUBIT names have changed due to an ACIS version change, then before importing the user should **set import mesh nodeset associativity off**.

Care should be taken that the geometry is merged the same way on export and import!

Between exporting and importing a mesh, the geometry may be modified slightly by compositing entities. But if entities are partitioned or a body is webcut, unless the new vertices match up almost exactly with nodes of the mesh, and the new curves match up almost exactly with edge chains of the mesh, etc., it will be impossible to associated the mesh with the geometry. If the user has trouble importing a mesh, he may import the mesh without associating it with any geometry by specifying the **preview** option. This puts the imported mesh entities in a group called ‘free_elements’. The user may then, e.g,, **draw fee_elements add** to see if the mesh indeed matches the geometry. Eventually, the user will want to get rid of these unassociated mesh elements by the **delete mesh preview** command.

If neither a **block** nor a **volume** is specified, then the entire mesh file is read. If a block is specified without specifying a volume, associativiy is used to determine which volume the block elements should be associated with. If a block and a volume are specified, the block elements are associated with the specified volume. If a volume is specified without a block, associativity data is used to find a block corresponding to the given volume.

Duplicating mesh

If the geometry to be meshed is similar to another meshed body, the user can use the command:

Copy Mesh Curve <curve_id_range> Onto Curve <curve_id_range>

[Source Node <starting node id> <ending node id>]

[Source Percent [<percentage>|auto]]

[Source [combine|SEPARATE]] [Target [combine|SEPARATE]]

[Source Vertex <id_range>] [Target Vertex <id_range>]

Copy Mesh Surface <surface_id> Onto Surface <surface_id>

[Source Face <id_range>]

[Source Node <id> Target Node <id>]

[Source Edge <id> Target Edge <id>]

[Source Vertex <id> Target Vertex <id>]

[Source Curve <id> Target Curve <id>] [Nosmoothing]

Copy Mesh Volume <volume_id> Onto Volume <volume_id>

[Source Vertex <vertex_id> Target Vertex <vertex_id>

[Source Curve <curve_id> Target Curve <curve_id>] [Nosmoothing]

For a discussion of this command, see the **Scheme Copy** description. The only difference between the scheme and this command is that this command takes place immediately.

▼ Mesh Quality Assessment

The ‘quality’ of a mesh can be assessed using several element quality metrics available in CUBIT. Online information about the CUBIT quality metircs can be obtained from the command

Quality Describe { hexahedral | tetrahedral | quadrilateral | triangular }

which gives data on the quality metrics for each of the above element types.

Metrics for Triangular Elements

For example, **quality describe triangular** yields the following information about CUBIT triangle metrics:

Table 5-4: Description of Triangular Quality Measures

Function Name	Dimension	Full Range	Acceptable Range	Reference
Aspect Ratio Gam	L^0	1 to inf	1 to 1.3	1

Table 5-4: Description of Triangular Quality Measures

Function Name	Dimension	Full Range	Acceptable Range	Reference
Element Area	0 to inf	None	None	2
Maximum Angle	degrees	60 to 180	60 to 90	2
Minimum Angle	degrees	0 to 60	30 to 60	2
Condition No.	L^0	1 to inf	1 to 1.3	3
Scaled Jacobian	L^0	-1.155 to +1.155	0.5 to 1.155	3

Triangular Quality Definitions:

Aspect Ratio Gamma: srms**2 / (2.30940108*area)

where Srms = sqrt(Sum(Si**2)/3), Si = edge length

Element Area: (1/2) * Jacobian at corner node

Maximum Angle: Maximum included angle in triangle

Minimum Angle: Minimum included angle in triangle

Condition No. Condition number of the Jacobian matrix at any corner

Scaled Jacobian: Minimum Jacobian divided by the lengths of 2 edge vectors

References for Triangular Quality Measures:

1. V. N. Parthasarathy et al, A comparison of tetrahedron quality measures, Finite Elem. Anal. Des., Vol 15(1993), 255-261.

2. Traditional.

3. P. Knupp, Achieving Finite Element Mesh Quality via Optimization of the Jacobian Matrix Norm and Associated Quantities, Part I, Int. J. Num. Meth. Engr.. 2000

Metrics for Quadrilaterals

Quality describe quadrilateral yields

Table 5-5: Description of Quadrilateral Quality Measures

Function Name	Dimension	Full Range	Acceptable Range	Reference
Aspect Ratio	L^0	1 to inf	1 to 4	1
Skew	L^0	0 to 1	0 to 0.5	1
Taper	L^0	0 to inf	0 to 0.7	1
Warpage	L^0	0 to inf	0 to 0.1	1
Element Area	L^2	-inf to +inf	None	1
Stretch	L^0	0 to 1	0.25 to 1	2
Minimum Angle	degrees	0 to 90	45 to 90	3
Maximum Angle	degrees	90 to 360	90 to 135	3
Oddy	L^0	0 to inf	0 to 16	4 & 5
Condition No.	L^0	1 to inf	1 to 4	5
Jacobian	L^2	- inf to inf	None	5
Scaled Jacobian	L^0	-1 to +1	0.5 to 1	5

Quadrilateral Quality Definitions:

Aspect Ratio: Maximum edge length ratios at quad center

Skew: Maximum |cos A| where A is the angle between edges at quad center

Taper: Maximum ratio of lengths derived from opposite edges

Warpage: Deviation of element from planarity.

Element Area: Jacobian at quad center

Stretch: Sqrt(2) * minimum edge length / maximum diagonal length

Minimum Angle: Smallest included quad angle (degrees).

Maximum Angle: Largest included quad angle (degrees).

Oddy: General distortion measure based on left Cauchy-Green Tensor

Condition No. Maximum condition number of the Jacobian matrix at 4 corners

Jacobian: Minimum pointwise volume of local map at 4 corners & center of quad

Scaled Jacobian: Minimum Jacobian divided by the lengths of the 2 edge vectors

References for Quadrilateral Quality Measures:

1. J. Robinson, CRE Method of element testing and the Jacobian shape parameters, Eng. Comput., Vol 4, 1987.

2. FIMESH code.

3. Unknown.

4. A. Oddy, J. Goldak, M. McDill, M. Bibby, A distortion metric for isoparametric finite elements, Trans. CSME, No. 38-CSME-32, Accession No. 2161, 1988.

5. P. Knupp, Achieving Finite Element Mesh Quality via Optimization of the Jacobian Matrix Norm and Associated Quantities, Part I, Int. J. Num. Meth. Engr.. 2000

Metrics for Tetrahedral Elements

Quality describe tetrahedral yields

Table 5-6: Description of Tetrahedral Quality Measures

Function Name	Dimension	Full Range	Acceptable Range	Reference
Aspect Ratio Bet	L^0	1 to inf	1 to 3	1
Aspect Ratio Gam	L^0	1 to inf	1 to 3	1
Element Volume	L^3	-inf to +inf	None	1
Condition No.	L^0	1 to inf	1 to 3	2
Jacobian	L^3	-inf to +inf	None	2
Scaled Jacobian	L^0	-1.414 to +1.414	0.5 to +1.414	2

Tetrahedral Quality Definitions:

Aspect Ratio Beta: CR / (3.0 * IR) where CR = circumsphere radius,

IR = inscribed sphere radius

Aspect Ratio Gamma: Srms**3 / (8.479670*V)

where Srms = sqrt(Sum(Si**2)/6), Si = edge length

Element Volume: (1/6) * Jacobian at corner node

Condition No. Condition number of the Jacobian matrix at any corner

Jacobian: Minimum pointwise volume at any corner

Scaled Jacobian: Minimum Jacobian divided by the lengths of 3 edge vectors

References for Tetrahedral Quality Measures:

1. V. N. Parthasarathy et al, A comparison of tetrahedron quality measures, Finite Elem. Anal. Des., Vol 15(1993), 255-261.
2. P. Knupp, Achieving Finite Element Mesh Quality via Optimization of the Jacobian Matrix Norm and Associated Quantities, Part II, Int. J. Numer. Meth. Engr., 2000

Metrics for Hexahedral Elements

Quality describe hexahedral yields

Table 5-7: Description of Hexahedral Quality Measures

Function Name	Dimension	Full Range	Acceptable Range	Reference
Aspect Ratio	L^0	1 to inf	1 to 4	1
Skew	L^0	0 to 1	0 to 0.5	1
Taper	L^0	0 to +inf	0 to 0.4	1
Element Volume	L^3	- inf to inf	None	1
Stretch	L^0	0 to 1	0.25 to 1	2
Diagonal Ratio	L^0	0 to 1	0.65 to 1	3
Dimension	L^1	0 to inf	None	1
Oddy	L^0	0 to inf	0 to 20	4,5
Condition No.	L^0	1 to inf	1 to 8	5

Table 5-7: Description of Hexahedral Quality Measures

Function Name	Dimension	Full Range	Acceptable Range	Reference
Jacobian	L^3	- inf to inf	None	5
Scaled Jacobian	L^0	-1 to +1	0.5 to 1	5

Hexahedral Quality Definitions:

Aspect Ratio: Maximum edge length ratios at hex center.

Skew: Maximum |cos A| where A is the angle between edges at hex center.

Taper: Maximum ratio of lengths derived from opposite edges.

Element Volume: Jacobian at hex center.

Stretch: Sqrt(3) * minimum edge length / maximum diagonal length.

Diagonal Ratio: Minimum diagonal length / maximum diagonal length.

Dimension: Pronto-specific characteristic length for stable time step calculation. Char_length = Volume / 2 grad Volume.

Oddy: General distortion measure based on left Cauchy-Green Tensor.

Condition No. Maximum condition number of the Jacobian matrix at 8 corners.

Jacobian: Minimum pointwise volume of local map at 8 corners & center of hex.

Scaled Jacobian: Minimum Jacobian divided by the lengths of the 3 edge vectors.

References for Hexahedral Quality Measures:

1. L.M. Taylor, and D.P. Flanagan, Pronto3D - A Three Dimensional Transient Solid Dynamics Program, SAND87-1912, Sandia National Laboratories, 1989.
2. FIMESH code
3. Unknown
4. A. Oddy, J. Goldak, M. McDill, M. Bibby, A distortion metric for isoparametric finite elements, Trans. CSME, No. 38-CSME-32, Accession No. 2161, 1988.
5. P. Knupp, Achieving Finite Element Mesh Quality via Optimization of the Jacobian Matrix Norm and Associated Quantities, Part II, Int. J. Num. Meth. Engr., 2000

Details on Robinson Metrics for Quadrilaterals

The quadrilateral element quality metrics that are calculated are aspect ratio, skew, taper, warpage, element area, and stretch. The calculations are based on metrics described in [18]. An illustration of the shape parameters is shown in Figure 5-30 The warpage is calculated as the Z

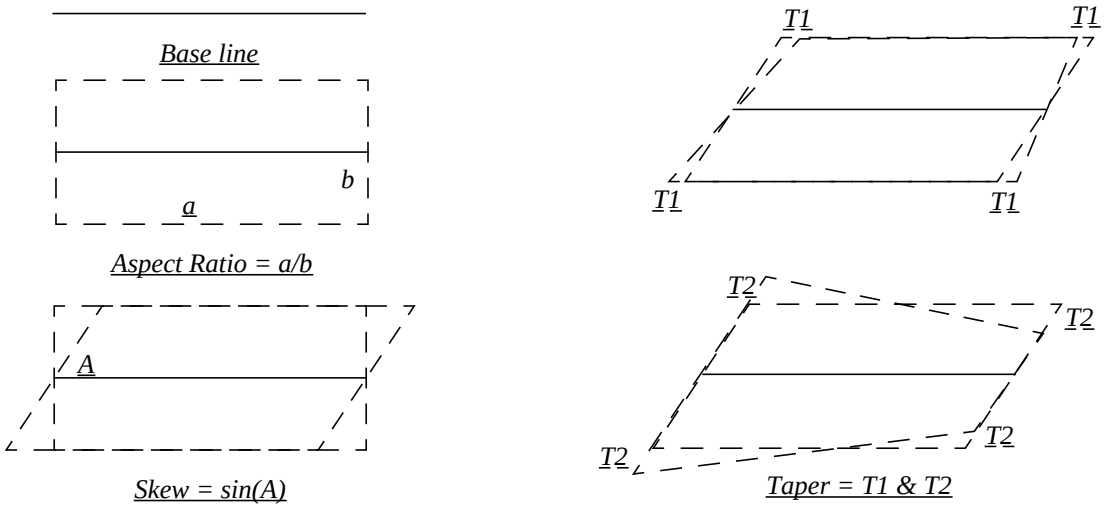


Figure 5-30: Illustration of Quadrilateral Shape Parameters (Quality Metrics)

deviation from the 'best-fit' plane containing the element divided by the minimum of 'a' or 'b' from Figure 5-30. The stretch metric is calculated by dividing the length of the shortest element edge divided by the length of the longest element diagonal.

Command Syntax

The base command to view the quality of a mesh is the following:

Quality {geom_and_mesh_list} [metric name] [quality options] [filter options]

Where the list contains surfaces and volumes and groups that have been meshed with faces, triangles, hexes, and tetrahedra; the list can also specify individual mesh entities or ranges of mesh entities.

If a specific **metric name** is given, only that metric or metrics are computed for the specified entities. Note that the metric given must be one which applies to the given entities; quality metrics and which entities to which they apply are summarized above.

The quality options are:

• [Global | Individual]:

If the user specifies **individual**, one quality summary is generated for each entity specified on the command line. If the user specifies **global**, or specifies neither, then one quality summary is generated for each mesh element type.

• [Draw [Histogram] [Mesh] [Monochrome] [Add]]:

If the user specifies **draw histogram**, then histograms are drawn, in a separate graphics window. The window contains one histogram for each quality metric. If the user specifies **draw mesh,** then the mesh elements are drawn in the default graphics window. The histogram and mesh graphics are color coded by quality: a smallest metric value corresponds to red, a largest metric value to blue, and in-between values according to the rainbow. If **monochrome** is specified, then the graphics are not color coded. If **add** is specified, then the current display is not cleared before drawing the mesh elements.

• [List [Detail] [Id] [Verbose Errors]]:

If the user specifies **List,** then the quality data is summarized in text form. **List Detail** lists the mesh elements by ascending quality metric. **List Id** lists the ids of the mesh elements. If **Verbose Errors** is specified, then details about unacceptable quality elements are printed out above the summaries.

There are several options available to filter the output of the quality command, using the following **filter options**:

• [High <value>] [Low <value>]:

Discards elements with metric values above or below value; either or both can be used to get elements above or below a specified value or in a specified range.

• [Top <number>] [Bottom <number>]:

Keeps only number elements with the highest or lowest metric values. For example, "**Quality hex all top 10**" would request the elements with the 10 highest values of the aspect ratio metric.

Example Output

The typical summary output from the command **quality surface 24** is shown in Table 5-8. Figure 5-31 left shows the corresponding histogram. The colored element display resulting from the command **quality surface 1 draw 'Skew'** is shown in Figure 5-31 right. A color legend is also printed to the terminal; see Table 5-9.

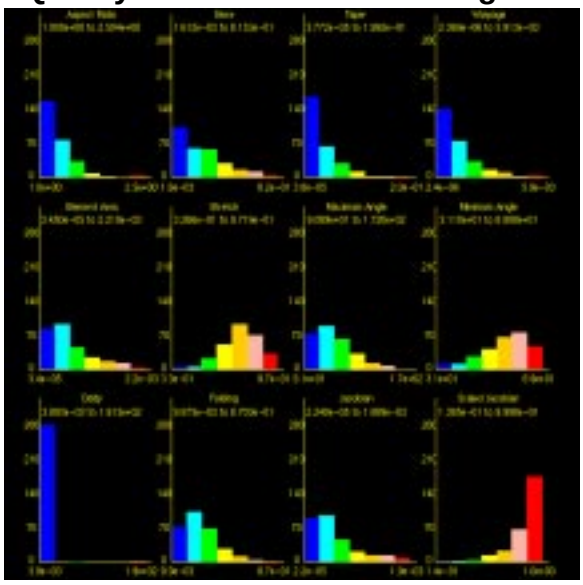
Table 5-8: Typical Summary for a Quality Command

```
Surface 24 Quad quality, 280 elements:
-------------------------------------
   Function Name    Average      Std Dev      Minimum   (id)    Maximum  (id)
 ---------------    ---------    ---------    --------------    -------------
    Aspect Ratio    1.257e+00    2.574e-01    1.000e+00 (2)  2.504e+00 (346)
           Skew     2.247e-01    1.808e-01    1.612e-03 (190)  8.153e-01 (80)
          Taper     3.076e-02    3.456e-02    3.772e-05 (259)  1.992e-01 (349)
        Warpage     1.199e-03    1.121e-03    2.369e-06 (271)  5.913e-03 (380)
    Element Area    6.335e-04    4.724e-04    3.450e-05 (15)  2.219e-03 (329)
         Stretch    7.406e-01    1.174e-01    3.266e-01 (156)  9.719e-01 (184)
   Maximum Angle    1.137e+02    1.509e+01    9.089e+01 (184)  1.720e+02 (112)
   Minimum Angle    6.835e+01    1.278e+01    3.110e+01 (80)  8.888e+01 (184)
            Oddy    2.025e+00    1.159e+01    3.893e-03 (183)  1.913e+02 (112)
         Folding    2.314e-01    1.447e-01    9.875e-03 (184)  8.703e-01 (112)
        Jacobian    5.195e-04    4.107e-04    2.240e-05 (14)  1.889e-03 (212)
 Scaled Jacobian    8.731e-01    1.353e-01    1.385e-01 (112)  9.998e-01 (184)
```

Table 5-9: Legend for Quality Surface 1 Skew Draw Mesh

```
Surface 24 Quad quality, 280 elements:
            Skew ranges from 1.612e-03 to 8.153e-01 (280 entities)
            Blue ranges from 1.612e-03 to 1.178e-01 (102 entities)
            Cyan ranges from 1.178e-01 to 2.341e-01 (60 entities)
           Green ranges from 2.341e-01 to 3.503e-01 (58 entities)
          Yellow ranges from 3.503e-01 to 4.666e-01 (29 entities)
       DkYellow ranges from 4.666e-01 to 5.828e-01 (15 entities)
            Pink ranges from 5.828e-01 to 6.990e-01 (12 entities)
             Red ranges from 6.990e-01 to 8.153e-01 (4 entities)
```

Quality Surface 24 Draw Histogram



Quality Surface 24 Skew Draw Mesh

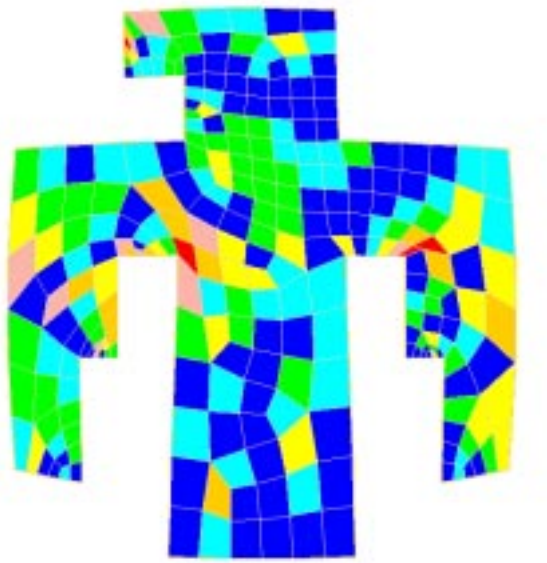


Figure 5-31: Illustration of Quality Metric Graphical Output

Controlling Mesh Quality

After a model is meshed, if the quality of mesh isn't acceptable, then there are two options available to improve that quality. The user can ask for more smoothing, or delete the mesh and start over. There are some tools that the user can invoke before meshing the model which can help to improve mesh quality. Skew control is one of these tools, and will be discussed here.

The philosophy behind the skew control algorithm is one of subdividing surfaces into blocky, four-sided areas which can be easily mapped. The goal of this subdivide-and-conquer routine is to lessen the skew that a mesh exhibits on submapped regions. By controlling the skew on these surfaces, the mesh of the underlying volume will also demonstrate less skew.

The commands for skew control are:

Control Skew Surface {surface_list} [Individual]

Delete Skew Control {surface_list} [Propagate]

The keyword **Individual** is deprecated. It's purpose is to specify that surfaces should be processed without regards to the other surfaces in the given list. This is not necessary, and could lead to problems with the final mesh. When the command is entered, the algorithm immediately processes the surfaces, inserting vertices and setting interval constraints on the resulting

subdivided curves. In this way the mesh is more constrained in its generation, and the resulting skew on the model can be lessened. The only surfaces which can utilize this algorithm are those which lend themselves to a structured meshing scheme, although future releases might lessen this restriction.

The user also has the ability to delete the changes that the skew control algorithm has made. This is done by using the **delete skew control** command. When the user requests the deletion of the skew control changes on a given surface, every curve on that surface will have the skew control changes deleted, even if a given curve is shared with another surface on which skew control was performed. If the user wishes to propagate the deletion of skew control to all surfaces which are affected by one (or more) particular surfaces, the keyword **propagate** should be used.

▼ Mesh Validity

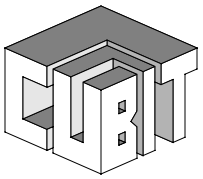
After a mesh is generated, it is checked to ensure that the mesh has valid connectivity. If an invalid mesh is formed, then CUBIT automatically deletes it. This default behavior can be changed with the following command:

Set Keep Invalid Mesh [on|off]

The current behavior can be viewed with the following command:

List Keep Invalid Mesh

The Jacobian quality metric is also computed automatically to check quality after a mesh is generated. If the quality is poor, a warning is printed to the terminal.



Chapter 6: Finite Element Model Definition and Output

▼ Introduction

This chapter describes the techniques used to complete the definition of the finite element model and the commands to export the finite element mesh to an Exodus database file. The definitions of the basic items in an Exodus database are briefly presented, followed by a description of the commands a user would typically enter to produce a customized finite element problem description.

▼ Finite Element Model Definition

Sandia's finite element analysis codes have been written to transfer mesh definition data in the ExodusII file format [6]. The ExodusII database exported during a CUBIT session is sometimes referred to as a Genesis database file; this term is used to refer to a subset of an Exodus file containing the problem definition only, i.e., no analysis results are included in the database.

The ExodusII database contains mechanisms for grouping elements into Element Blocks, which are used to define material types of elements. ExodusII also allows the definition of groups of nodes and element sides in Nodesets and Sidesets, respectively; these are useful for defining boundary and initial conditions. Using Element Blocks, Nodesets and Sidesets allows the grouping of elements, nodes and sides for use in defining boundary conditions, without storing analysis code-specific boundary condition types. This allows CUBIT to generate meshes for many different types of finite element codes.

Element Blocks

Element Blocks (also referred to as simply, *Blocks*) are a logical grouping of *elements* all having the same basic geometry and number of nodes. All elements within an Element Block are required to have the same element type. Access to an Element Block is accomplished through a

user-specified integer Block ID. Typically, Element Blocks are used by analysis codes to associate material properties and/or body forces with a group of elements.

Nodesets

Nodesets are a logical grouping of *nodes* accessed through a user-specified Nodeset ID. Nodesets provide a means to reference a group of nodes with a single ID. They are typically used to specify load or boundary conditions on portions of the CUBIT model or to identify a group of nodes for a special output request in the finite element analysis code.

Sidesets

Sidesets are another mechanism by which constraints may be applied to the model. Sidesets represent a grouping of *element sides* and are also referenced using an integer Sideset ID. They are typically used in situations where a constraint must be associated with element sides to satisfactorily represent the physics (for example, a contact surface or a pressure.

Element Types

The basic elements used to discretize geometry were described in a previous chapter (see “Element Types” on page 180). Within each basic element type, several specific element types are available; these specific element types vary by the number of nodes used to define the element, and result in different orders of accuracy of the element. The element types available for each basic element type defined in CUBIT are summarized in Table 6-1. For a description of the node and side numbering conventions for each specific element type, see Appendix D.

Element types can be set for individual Element Blocks, either before or after meshing has been performed. Higher-order nodes are created only when the mesh is being exported to the ExodusII file, and persist in the CUBIT database after file export.

Table 6-1: Element types defined in CUBIT.

Basic Element Type	Specific Element Types	Notes
`Edge`	BAR, BEAM	Bars have 2 DOF’s per node, Beams 3
`Triangle`	TRI, TRI3, TRI6, TRI7	Tri element nodal coordinates are always 3D.
`Quadrilateral`	QUAD, QUAD4, QUAD8, QUAD9; SHELL, SHELL4, SHELL8, SHELL9	Quad element nodal coordinates are 2D, that is their nodes contain only x and y coordinates. Shell element nodal coordinates are 3D.
`Tetrahedron`	TETRA, TETRA4, TETRA8, TETRA10	TETRA8 contains vertex nodes and mid-face nodes, experimental element used in Sandia FEA research
`Hexahedron`	HEX, HEX8, HEX20, HEX27	

Element Block Specification

Element blocks are the method CUBIT uses to group related sets of elements into a single entity. Each element in an element block must have the same basic and specific element type. Element Blocks may be defined for volumes, surfaces, and curves. Element blocks are defined with the following Block commands.

Block <block_id> {Curve | Surface | Volume} <range>

Block <block_id_range> Element Type <type>

Block <block_id_range> Attribute <value>

Some important notes regarding Element Blocks are as follows:

- Multiple volumes, surfaces, and curves can be contained in a single element block
- A volume, surface, or curve can only be in one element block
- Element Block id's are arbitrary and user-defined. They do not need to be in any contiguous sequence of integers.
- Element Blocks can be assigned a single floating point number, referred to as the block Attribute; this number is used to represent the length or thickness of Bar and Shell elements, respectively. The attribute defaults to 1.0 if not specified.

When exporting an ExodusII file, if the user has not specified any Element Blocks, by default element blocks will be written for any meshed volumes. This default behavior can be changed, to write surface, volume, or no meshes by default. This option can be set using the command

Set Default Blocks [on|off|Volume|Surface]

If the element type is not assigned for an element block, it will be assigned a default type depending on which type of geometry entity is contained in the block; default element block id's are also determined by the geometry entity being meshed. The default values used for element type and id are:

Volume: The default block ID will be set to the Volume ID and 8-node hexahedral elements will be generated.

Surface: The block ID will be set to 0 and 4-node shell elements will be generated.

Curve: The block ID will be set to 0 and 2-node bar elements will be generated.

Several examples of specifying various types of element blocks are given in Appendix A:.

Nodesets and Sidesets

Boundary conditions such as constraints and loads are applied to the finite element model using nodesets and sidesets. Nodesets can be created from groups of nodes categorized by their owning volumes, surfaces, or curves. Nodes can belong to more than one nodeset. Sidesets can be created from groups of element sides or faces categorized by their owning surfaces or curves. Element sides and faces can belong to more than one sideset. Nodesets and Sidesets can be viewed individually through CUBIT by employing the **Draw Nodeset** and **Draw Sideset** commands.

Nodesets and Sidesets may be assigned to the appropriate geometric entities in the model using the following commands:

Nodeset <nodeset_id> {Curve | Surface | Volume | Vertex} <range>

Sideset <sideset_id> Surface <range>

[Remove|Forward|Reverse|Both|wrt Volume <id>]

Sideset <sideset_id> Curve <range> [Remove|wrt {Surface <id>|All}]

Like element blocks, Nodesets and Sidesets are given arbitrary, user-defined ID numbers. If there are no user-defined Nodesets or Sidesets, none are written to the ExodusII file.

With Sidesets, direction is often important. For surfaces, the direction may be specified using the **Forward**, **Reverse**, or **Both** options. The **Forward** option will write a sideset for the hexes in the surface's forward volume, which is the volume that the surface's normal points away from. The **Reverse** option will write a sideset for the hexes in the surface's reverse volume, which is the volume that the surface's normal points into. The **Both** option will allow sidesets to be written for the hexes that lie in volumes on both sides of the surface. The default is **Forward**. The user can additionally specifiy the volume from which the hexes should be taken by the **wrt Volume** option.

Direction is equally important for curves in Sidesets. The **wrt Surface** option allows the user to indicate which surface's faces will be included in the Sideset. The **wrt All** option will include all faces attached to the curve. The default is **wrt All**.

Nodeset Associativity Data

Nodesets can be used to store geometry associativity data in the ExodusII file. This data can be used to associate the corresponding mesh to an existing geometry in a subsequent CUBIT session. This functionality can be used either to associate a previously-generated mesh with a geometry ("Mesh Importing and Duplicating" on page 167), or to associate a field function with a geometry for adaptive surface meshing (see "Adaptive Meshing" on page 215).

The commands to control and list whether associativity data is written or read from an ExodusII files are the following. Note that **Complete** is only used for Adaptive Meshing, while the other options are useful for re-importing meshes into CUBIT.

List Import Mesh NodeSet Associativity

List [Export Mesh] NodeSet Associativity

List [Export Mesh] NodeSet Associativity Complete

set Import Mesh NodeSet Associativity [On|Off]

set [Export Mesh] NodeSet Associativity [On|Off]

set [Export Mesh] NodeSet Associativity Complete [On|Off]

Associativity data is stored in the ExodusII file in two locations. First, a nodeset is written for each piece of geometry (vertices, curves, etc) containing the nodes owned for that geometry. Then, the name of each geometry entity is associated with the corresponding nodeset by writing a property name and designating the corresponding nodeset as having that property. Nodeset numbers used for associativity nodesets are determined by adding a fixed base number (depending on the order of the geometric entity) to the geometric entity id number. The base

numbers for various orders of geometric entities are shown in Table 6-2. For example, nodes owned by curve number 26 would be stored in associativity nodeset 40026.

Table 6-2: Nodeset id base numbers for geometric entities

Geometric Entity	Base Nodeset Id
Vertex	50000
Curve	40000
Surface	30000
Volume	20000

Instead of storing just the nodes owned by a particular entity, nodes for lower order entities are also stored. For example, the associativity nodeset for a surface would contain all nodes owned by that surface as well as the nodes on the bounding curves and vertices.

ExodusII Model Title

CUBIT will automatically generate a default title for the Genesis database. The default title has the form:

cubit(genesis_filename): date: time

The title can be changed using the command:

Title '<title_string>'

Transforming Mesh Coordinates

A mesh can be transformed to a new location as it is written to an Exodus file. To transform a mesh during export use the following command:

Transform Mesh Output
[Scale <factor> [<factor> <factor>]]
[Scale {X|Y|Z} <factor>]
[Translate <dx> [<dy> [<dz>]]]
[Translate {X|Y|Z} <distance>]
[Rotate <degrees> about {X|Y|Z}]
[Reset]

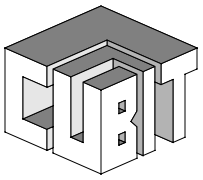
This command may be repeated any number of times using any number of options. Transform commands are cumulative, added to the effect of previous transforms. Use the **Reset** option to clear the transformation matrix and to prevent previous transformation commands from affecting node positions in the output file.

Transforming a mesh during output does not change the position of the mesh within CUBIT. It only changes the nodal positions written to the Exodus file.

Exporting the Finite Element Model

After defining the element blocks, nodesests and sidesets for a model, the model can be written to the ExodusII file using the command:

Export Genesis '<filename>'



References

1 T. D. Blacker and M. B. Stephenson, 'Paving: a new approach to automated quadrilateral mesh generation', SAND90-0249, Sandia National Laboratories, (1990).

2 M. B. Stephenson, S. A. Canann, and T. D. Blacker, 'Plastering: a new approach to automated, 3D hexahedral mesh generation', SAND89-2192, Sandia National Laboratories, (1992).

3 G. D. Sjaardema, et. al., *CUBIT Mesh Generation Environment, Volume 2: Developers Manual*, SAND94-1101, Sandia National Laboratories, (1994).

4 Spatial Technology, Inc., *ACIS Test Harness Application Guide Version 1.4*, Spatial Technology, Inc., Applied Geometry, Inc., and Three-Space, Ltd., (1992).

5 T. D. Blacker, *FASTQ Users Manual Version 1.2*, SAND88-1326, Sandia National Laboratories, (1988).

6 L. A. Schoof, *EXODUS II Application Programming Interface*, internal memo, Sandia National Laboratories, (1992).

7 W. A. Cook and W. R. Oakes, 'Mapping methods for generating three-dimensional meshes', *Comp. mech. eng.*, **Volume 1**, 67-72 (1982).

8 R. E. Jones, *QMESH: A Self-Organizing Mesh Generation Program*, SLA - 73 - 1088, Sandia National Laboratories, (1974).

9 R. E. Tipton, 'Grid Optimization by Equipotential Relaxation', unpublished, Lawrence Livermore National Laboratory, (1990).

10 A. P. Gilkey and G. D. Sjaardema, *GEN3D: A GENESIS Database 2D to 3D Transformation Program*, SAND89-0485, Sandia National Laboratories, (1989).

11 G. D. Sjaardema, *GREPOS: A GENESIS Database Repositioning Program*, SAND90-0566, Sandia National Laboratories, (1990).

12 G. D. Sjaardema, *GJOIN: A Program for Merging Two or More GENESIS Databases*, SAND92-2290, Sandia National Laboratories, (1992).

13 G. D. Sjaardema, *APREPRO: An Algebraic Preprocessor for Parameterizing Finite Element Analyses*, SAND92-2291, Sandia National Laboratories, (1992).

14 G. D. Sjaardema, *Overview of the Sandia National Laboratories Engineering Analysis Code Access System*, SAND92-2292, Sandia National Laboratories, (1993).

15 S. C. Lovejoy and R. G. Whirley, *DYNA3D Example Problem Manual*, UCRL-MA--105259, University Of California and Lawrence Livermore National Laboratory, (1990).

16 Open Software Foundation, Inc., *OSF/Motif™ User's Guide Revision 1.2*, PTR Prentice Hall, Englewood Cliffs, New Jersey, (1993).

17 J. M. Osier, *Keeping Track, Managing Messages with GNATS, The GNU Problem Report Management System*, Users manual for GNATS Version 3.2, Cygnus Support, October 1993.

18 J. Robinson, "CRE method of element testing and Jacobian shape parameters, *Eng. Comput.*, Vol. 4 (1987).

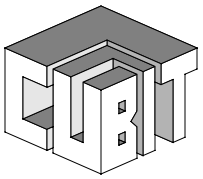
19 L. M. Taylor and D. P. Flanagan, Pronto 3D—A Three-Dimensional Transient Solid Dynamics Program, SAND87-1912, Sandia National Laboratories, (1989).

20 S. W. Attaway, unpublished, (1993).

21 A.Oddy, J. Goldak, M. McDill, and M. Bibby "A Distortion Metric for Isoparametric Finite Elements" Transactions of the Canadian Soc. Mech. Engr., pp213-217, Vol 12, No 4, 1988.

22 V. N. Parthasarathy, C.M. Graichen, A.F. Hathaway, "A comparison of tetrahedron quality measures", Finite Elem. Anal. Des., Vol 15(1993), 255-261.

CHAPTER



Appendix A: Examples

▼ Introduction

The purpose of this Appendix is to demonstrate the capabilities of CUBIT for finite element mesh generation as well as provide a few examples on the use of CUBIT. Some examples also demonstrate the use of the ACIS test harness as well as other related programs. This Appendix is not intended to be a step-by-step tutorial.

▼ General Comments

The examples in this appendix show the use of CUBIT under various scenarios. To reproduce these examples, the user would need the journal files containing the CUBIT commands described below, and in some cases an ACIS SAT file containing model geometry. The journal files and SAT files necessary for running these examples are available from the CUBIT web site. For examples not requiring SAT files, the user can also type in the commands described for that example.

The examples in this appendix each cover several of CUBIT's mesh generation capabilities. The CUBIT features exercised by each example are shown in Table A-1.

Examples	Geometry Features				Surface Meshing Features						Volume Meshing Features			
	Primitives	Booleans	Decomposition	Merging	Interval Assignment	Submapping	Mapping	Paving	Triangle Tool	Smoothing	Sweeping	Submapping	Mapping	Tetrahedron
Internal Geometry	x	x			x			x			x	x		
Sphere Octant	x	x	x	x	x			x	x	x	x	x		x
Box Beam	x			x	x		x							
Thunderbird	x	x			x			x						
Advanced Tutorial			x	x	x	x	x	x		x	x	x	x	

Table A-1: CUBIT Features Exercised by Examples.

▼ Simple Internal Geometry Generation

This simple example demonstrates the use of the internal geometry generation capability within CUBIT to generate a mesh on a perforated block. The geometry for this case is a block with a cylindrical hole in the center. It illustrates the **brick**, **cylinder**, **subtract**, **pave**, and **translate** commands and boolean operations. The geometry to be generated is shown in Figure A-1. This figure also shows the curve and surface labels specified in the CUBIT journal file. The final meshed body is shown in Figure A-2. The CUBIT journal file is:

```
# Internal Geometry Generation Example
Brick Width 10. Depth 10. Height 10.              # Create Cube
Cylinder Height 12. Radius 3.     # Create cylinder through Cube
View From 15 20 25                        # Move to new Viewpoint
Display       #You may want to move to graphics window to mouse
                                  #around to get the feel for it
Subtract 2 From 1       # Remove cylinder from cube—create hole
Body 3 Size 1.0               # Default element size for model
Label Curve On
Label Surface On   #Turn on curve and surface labels for scheme
                                        #and size specification
Display
Surface 10 Interval 10   # Change intervals on cylinder surface
Curve 15 to 16 Interval 20 # Change intervals around cyl. circ.
Surface 11 Scheme Pave                     # Front surface paved
Volume 3 Scheme Sweep Source 11 Target 12            #Remainder
                                    # of block will be meshed by
                          # sweeping front surface to back surface
Mesh Volume 3                                  # Create the mesh
Graphics Mode Hiddenline # Hiddenline view of cube (Figure B-2)
```

The first two lines create a 10 unit cube centered at the origin and a cylinder with radius 3 units and height of 12 units also centered at the origin. The cylinder height is arbitrary as long as it is greater than the height of the brick. The **subtract** command then performs the boolean by subtracting the cylinder (body 2) from the block (body 1) to create the final geometry (body 3). The remainder of the commands simply assign the desired number of intervals and then generate the mesh. Note that since the cylindrical hole is a "periodic surface," there are no edges joining the two curves so the number of intervals along its axis must be set by the surface interval command.

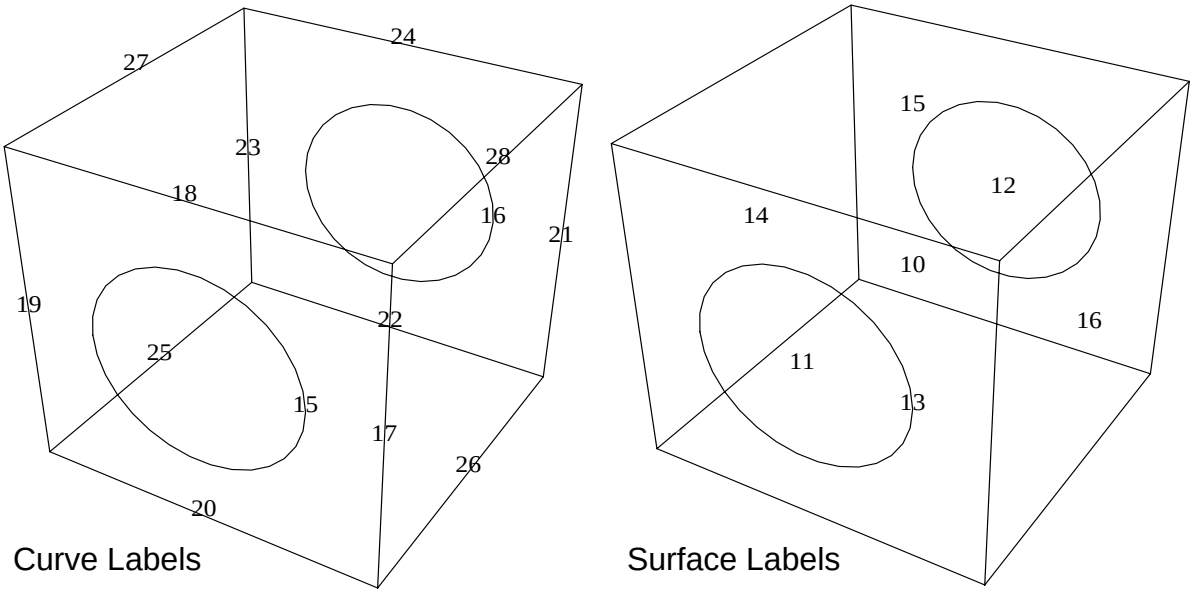


Figure A-1: Geometry for Cube with Cylindrical Hole

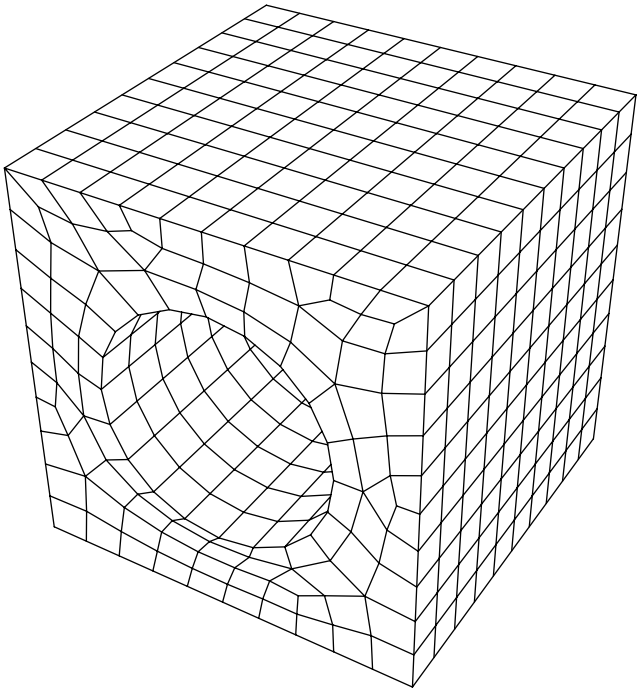


Figure A-2: Generated Mesh for Cube with Cylindrical Hole

Octant of Sphere

This example also illustrates the internal geometry generation capabilities of CUBIT to generate an octant of a sphere. The procedure used is to generate the octant by creating a sphere only on the positive quadrant of the reference frame. Two methods of meshing are demonstrated in this example: one is to decompose the octant into to volumes - a central “core” and an outer “peel” which are both meshable using the sweep schemes. The second is to mesh the octant with the triangle and tetrahedron meshing scheme. This example uses the **sphere, webcut, merge, auto, triangle, tetrahedron and smooth** commands.

The following annotated CUBIT journal file will generate the meshes shown in Figure A-3.

```
## create an octant of a sphere on the positive quadrant
Sphere Radius 10.0 xpos ypos zpos            #create the octant
Webcut Body 3 Cylinder Radius 4 Axis z Noimprint Nomerge

## coalesce redundant surfaces
Merge All

Volume 5 Size 0.4999
Volume 4 Size 0.6

Volume all Scheme Auto         #Use auto to set meshing schemes
List Volume 4               #List the volume to see the schemes
List Volume 5               #List the volume to see the schemes
Mesh Volume all

#now try it with the tetrahedron this way
Reset
Sphere Radius 10. xpos ypos zpos              #Create an octant
## the tetrahedron scheme will mesh a tetrahedron with hexes
Volume 3 scheme tetrahedron #mesh the volume with the primitive
Surface All Scheme Triangle   #Surfaces must be scheme triangle
Volume 3 size 0.7                          #Set an interval size
Mesh Surface all                        #First mesh the surfaces
Smooth Surface all    #Scheme Triangle often requires smoothing
Mesh Volume 3                               #Mesh the volume
Export Genesis’Octant.gen’                 #Write out the mesh
```

Box Beam

A simple example using CUBIT is the box beam buckling problem shown in Figure A-4. A description of an analysis which uses this type of mesh is found in [15]. This example uses the **merge**, **nodeset** and **block** commands and the mapping mesh generation scheme. The geometry is generated inside of CUBIT using Aprepro commands and variables. The geometry file is as follows:

 # File: boxBeamGeom.jou

```
# Side = {Side = 1.75}
# Height = {Height = 12.0}
# Upper = {Upper = 2.0}
Brick Width {Side/2.0} depth {Side/2.0} height {Height-Upper}
Body 1 name “lowerSection”
Brick Width {Side/2.0} depth {Side/2.0} height {Upper}
Body 2 name “upperSection”
Move lowerSection xyz {Side/4.0} {Side/4.0} {(Height-Upper)/
2.0}
Move upperSection xyz {Side/4.0} {Side/4.0} {Upper/2.0 + Height
```

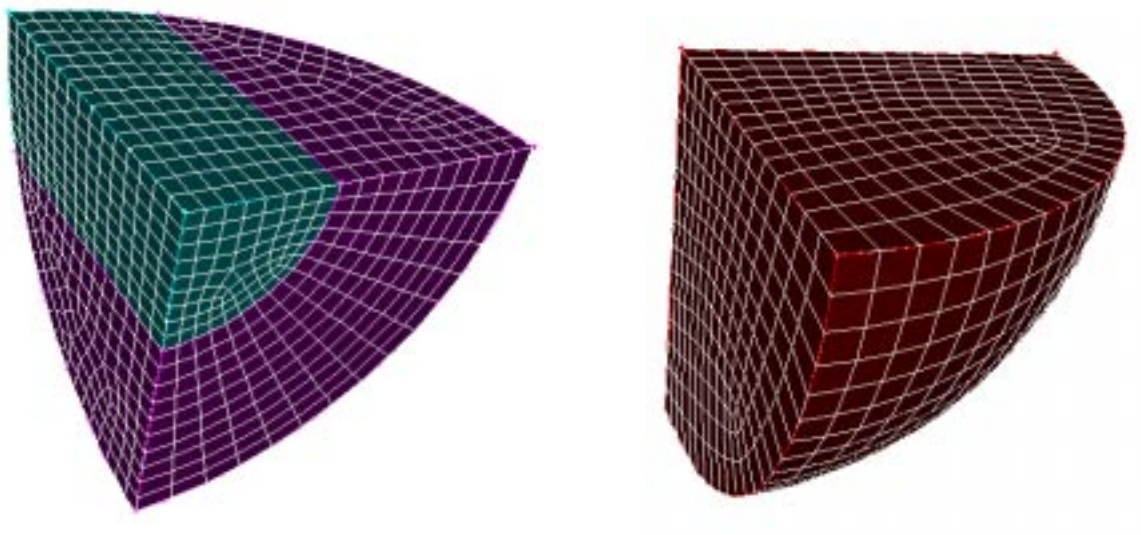


Figure A-3: Mesh for Octant of Sphere via Coring/Sweeping (left) and the Tetrahedron Primitive (Right)

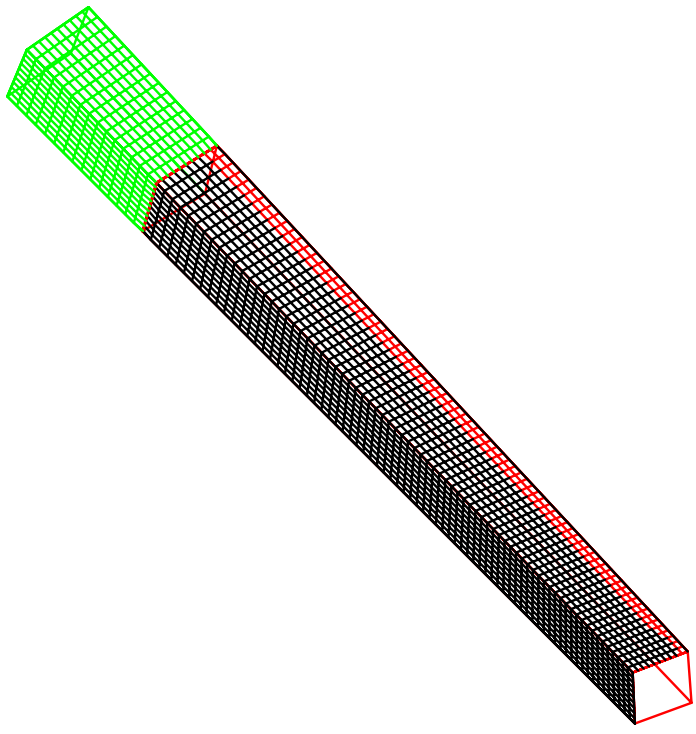


Figure A-4: Box Beam example

```
- Upper}
Export acis “boxBeam.sat”                    #Save the file to SAT
```

In this example, it is assumed that subsequent analyses will take advantage of the problem symmetry and therefore only one-quarter of the box beam will be meshed. It is worth noting that

there are a variety of ways to construct a solid model for this problem; however, experience thus far with ACIS and CUBIT indicates that the easiest way to model the box beam is to use ACIS block primitives[1]. Even though subsequent meshing will only be performed on the faces of the solid model, the entire 3D body is saved as an ACIS.sat file. The CUBIT journal file for the box beam example is:

 # File: boxBeam.jou

```
# Thickness = {Thickness = 0.06}
# Crease = {Crease = 0.01}
# XYInts = {XYInts = 10}
# ZInts = {ZInts = 90}
# UpperInts = {UpperInts = 15}

Import Acis 'boxBeam.sat'

Merge All
Label Surface on
Label Curve on
Display

Curve 1 To 8 Interval {XYInts}
Curve 13 To 16 Interval {XYInts}

Curve 9 To 12 Interval {ZInts-UpperInts}
Curve 21 To 24 Interval {UpperInts}

Mesh Surface 3
Mesh Surface 6
Mesh Surface 9
Mesh Surface 12

NodeSet 1 Curve 1
NodeSet 2 Curve 4

NodeSet 1 Move {-Crease} 0 0
NodeSet 2 Move 0 {Crease} 0

Block 2 Surface 3
Block 2 Surface 6

Block 1 Surface 9
Block 1 Surface 12

Block 1 To 2 Attribute {Thickness}

Export Genesis 'boxBeam.exoII'
Quit
```

Commands worth noting in the CUBIT journal file include:

- **Block, Block Attribute** Allows the user to specify that shell elements for the surfaces of the solid model are to be written to the output (EXODUSII) database, and that shell elements be given a thickness attribute. This is necessary since CUBIT defaults to three-dimensional hexahedral meshing of solid model volumes.

- **NodeSet Move** Allows the user to actually move the specified nodes by a vector (Δx, Δy, Δz). This is advantageous for the buckling problem, since the numerical simulation requires a small "crease" in the beam in order to perform well.

1. This geometry can also be generated using the internal CUBIT Brick primitive.

- **Merge** Allows the user to combine geometric features (e.g. edges and surfaces).

Other commands in the journal file should be straightforward. Since the problem is sufficiently simple to mesh using a mapping transformation, specification of a meshing "scheme" is unnecessary (mapping is the default in CUBIT).

Finally, note that both the CUBIT journal files (**boxBeamGeom.jou and boxBeam.jou**) contain macros that are evaluated using Aprepro. The *makefile* is used to semi-automatically generate the mesh is given below:

 # File: Makefile

```
boxBeam.g:boxBeam.exoII
    ex2exlv2 boxBeam.exoII boxBeam.g

boxBeam.exoII:boxBeam.sat boxBeam.jou
    cubit -batch -nographics boxBeam.jou

boxBeam.sat: boxBeamGeom.jou
    cubit -batch -nographics boxBeamGeom.jou

boxBeam.jou: boxBeam.jou

clean:
    @-rm *.sat *.exoII *.g
```

While this particular example is a trivial use of the software, it does serve to demonstrate a few of the capabilities offered by CUBIT.

▼ Thunderbird 3D Shell

This example is the three-dimensional paving of a shell shown in Figure A-5. The 2D wireframe geometry of the thunderbird is given by the following FASTQ file:

#File: tbird.fsq

```
TITLE
MESH OF SANDIA THUNDERBIRD

$ block {e = .2}  int= {isq = 20}
$ number of elements in block thick {iblkt = 5 } block thickness
{blkt=.2 }
$ block angle {angle=15}
$ magnification factor = {magnificationFactor=1.0}
$ bird {bthick = .018} {ithick = 3} {idepth = 20}
$ {pi = 3.14159265359} {rad=magnificationFactor/pi} {bdepth=1.}
$ preferred normalized element size = {elementSize=0.06}
$ number of intervals along outside edges =
$ {border_int=5} {corner_int=10} {side_int=20}
$  {outsideIntervals= 2*corner_int+side_int}
$ {boxTop=.2} {topIntervals = 8}

$ {insideCurveInt=8}


$ {MAG=magnificationFactor/3.0}

$ {middleInside=MAG*0.97}
$ {xCurveStartInside=MAG*0.60}
$ {yCurveStartInside=MAG*0.93}
$ {curveMiddleInside=MAG*0.81}

$ {xCurveStartOutside=MAG*0.75}
$ {yCurveStartOutside=MAG*1.17}
$ {middleOutside=MAG*1.20}
$ {curveMiddleOutside=MAG*1.01}
$ {boundingBox = MAG*1.5}
```

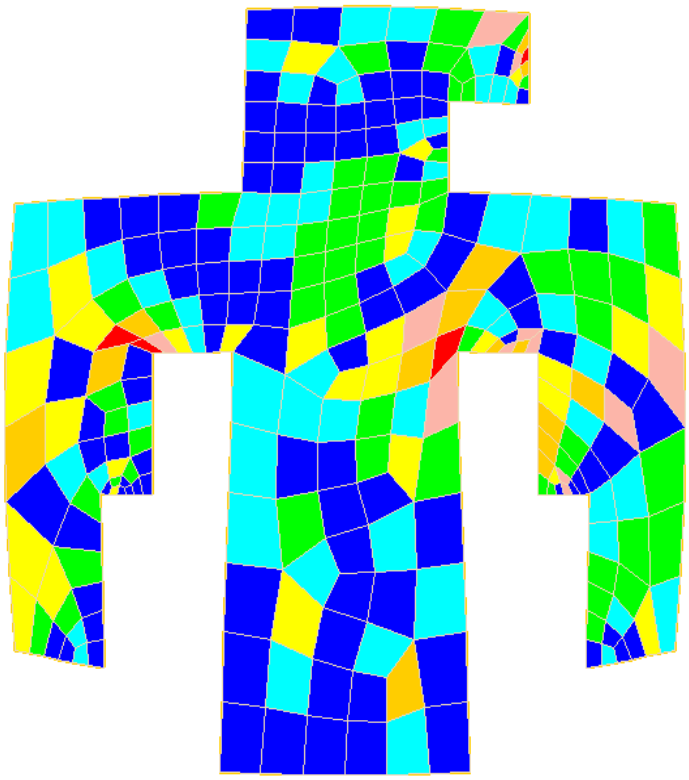


Figure A-5: Sandia Thunderbird 3D shell

```
$ Thunderbird Coordinates

POINT 1 {MAG*-.40} {MAG*.78}
POINT 2 {MAG*-.40} {MAG*.59}
POINT 3 {MAG*-.22} {MAG*.59}
POINT 4 {MAG*-.22} {MAG*.40}
POINT 5 {MAG*-.75} {MAG*.40}
POINT 6 {MAG*-.78} {MAG*-.09}
POINT 7 {MAG*-.75} {MAG*-.58}
POINT 8 {MAG*-.53} {MAG*-.60}
POINT 9 {MAG*-.54} {MAG*-.23}
POINT 10 {MAG*-.42} {MAG*-.23}
POINT 11 {MAG*-.42} {MAG*.07}
POINT 12 {MAG*-.24} {MAG*.07}
POINT 13 {MAG*-.27} {MAG*-.80}
POINT 14 {MAG*.27} {MAG*-.80}
POINT 15 {MAG*.24} {MAG*.07}
POINT 16 {MAG*.42} {MAG*.07}
POINT 17 {MAG*.42} {MAG*-.23}
POINT 18 {MAG*.54} {MAG*-.23}
POINT 19 {MAG*.53} {MAG*-.60}
POINT 20 {MAG*.75} {MAG*-.58}
POINT 21 {MAG*.78} {MAG*-.09}
POINT 22 {MAG*.75} {MAG*.40}
POINT 23 {MAG*.22} {MAG*.40}
POINT 24 {MAG*.21} {MAG*.78}
POINT 25 {MAG*0.0} {MAG*.80}

$ lines for Tbird

LINE 1 STR 1 2
```

```
LINE 2 STR 2 3
LINE 3 STR 3 4
LINE 4 STR 4 5
LINE 5 CIRM 5 7 6
LINE 6 STR 7 8
LINE 7 STR 8 9
LINE 8 STR 9 10
LINE 9 STR 10 11
LINE 10 STR 11 12
LINE 11 STR 12 13
LINE 12 STR 13 14
LINE 13 STR 14 15
LINE 14 STR 15 16
LINE 15 STR 16 17
LINE 16 STR 17 18
LINE 17 STR 18 19
LINE 18 STR 19 20
LINE 19 CIRM 20 22 21
LINE 20 STR 22 23
LINE 21 STR 23 24
LINE 22 STR 24 1 0 7 1.0

$ REGIONS

SIZE {elementSize*MAG}

REGION 1 1 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 *
 -16 -17 -18 -19 -20 -21 -22

SCHEME 0 X
BODY 1
EXIT
```

A command interpreter has been developed inside CUBIT to convert FASTQ geometry into CUBIT's modeling system (ACIS). The previous file, tbird.fsq, can be read into CUBIT by the command:

Import Fastq "<file_name>"

The file can be read into CUBIT and converted from a 2D "sheet" body to a 3D solid, by the following commands:

#File: tbird3dGeom.jou

```
import fastq "tbird.fsq"
cylinder radius.5 height 1.25
rotate body 2 about x angle 90
sweep surface 1 vector 0 0 1 distance 1
intersect body 3 with body 2
export acis "tbird3d.sat"
```

This example shows a powerful technique of generating two dimensional surfaces through FASTQ or through CUBIT's own bottom up geometry creation, and then sweeping them into three dimensional shapes.

In this example, only the 3D shell of the thunderbird is desired for the finite element model, and thus, the block command is used to specify that only elements on the surface are to be created. The following CUBIT journal file demonstrates current 3D paving capability:

#File: tbird3d.jou

```
Import Acis 'tbird3d.sat'
#You may want to move the view around by mousing in the Graphics
#window to get a better idea of the 3D shaped surface.
Label Surface on
Display
Draw Surface 23
```

```
Draw Surface 24

Surface 24 Size 0.03
Surface 24 Scheme Pave
Mesh Surface 24
#Show the quality of mesh measured by skew (note the scale)
Quality surface 24 draw mesh skew

Block 1 Surface 24
Block 1 Attribute 0.03
Export Genesis “tbird.g2”
```

▼ Advanced Tutorial

The objective of this example is to illustrate the use of some advanced meshing operations to mesh a more complex geometry. The example purposely does not do everything right the first time to demonstrate the thought process a user would go through when meshing a real part for the first time. This example demonstrates the use of **webcut** to decompose the model into sweepable volumes, manually setting **meshing schemes** when **scheme auto** fails for certain volumes and **matching intervals** to ensure meshing scheme constraints are meet. It should be noted that the sequence of commands is important to successfully generate the meshed model. It is recommended that the user first perform all the decomposition on the model, then **imprint** the entire model. Imprinting ensures that the topology of adjacent bodies match so that correct merging of adjacent surfaces can be performed. Next, use the merge all command to merge the common surfaces and ensure a contiguous mesh throughout the model. It is important to watch the merge all command output, since during typical **merge all** operations, all of the curves and vertices will be merged during the surface merging. Thus unless specifically desired, curve and vertex merging messages should not be seen from this command. If these are reported during the execution of the command, it may indicate invalid topology (remedied by an **imprint all**) or some other invalidity in the model. Performing an **imprint all** after the **merge all** may corrupt the data base; the user should not perform geometry operations after the **merge** command. Next, set the element size (e.g. **volume all size 15**) then the meshing schema (e.g. **volume all scheme auto**). The regime is finished when the mesh command is issued. Setting up BC’s and **Element Blocks** are not covered in this tutorial.

The command set default names on assigns names to the geometric entities. These names are saved with the geometry when the file is saved and also remain constant within code revisions. Throughout the session, each entity will acquire multiple names and any name given for each entity is valid for identification.

The ACIS SAT file for this tutorial can be obtained via the Cubit website at:

http://endo.sandia.gov/cubit/turorial_files.html

The geometry used in this example is shown in Figure A-6. The journal file for this exercise is given below as follows. The resulting mesh is shown in Figure A-7.

```
## Turn on the default names so when the file is read in,
## the entities will be named.
## Assign names to geometric entities such as cur01, surf10, ##
## etc.
## read in the file
## Some default names have already been set in the sat file.
## Turn on default names after the original has been read in
## so only entities created after the import can be read.
import acis “advanced_demo.sat”
```

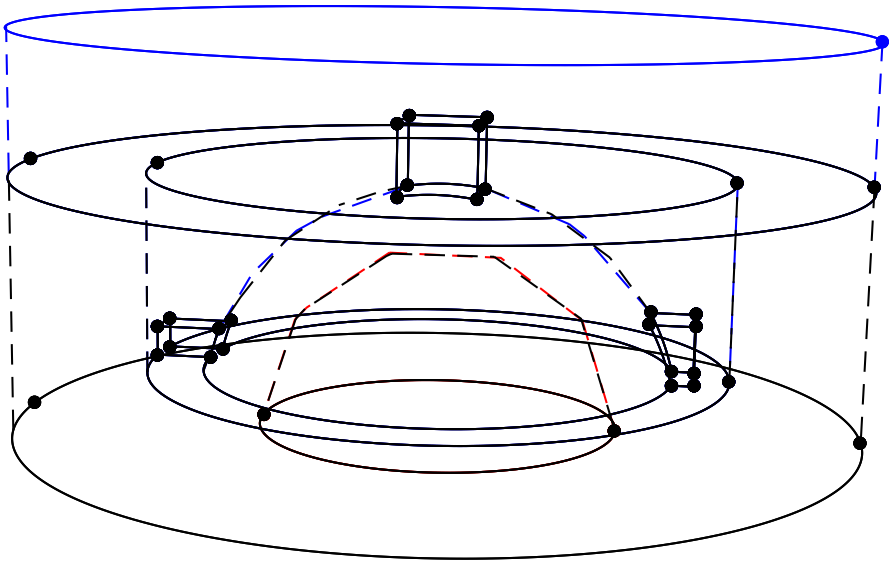


Figure A-6: Geometry of Advanced Tutorial

```
## Set a better graphics mode for viewing.
graphics mode smoothshade
## Look at each body to get a feel for what needs to occur.
draw body 1
draw body 2
draw body 3
## Make sure we do fast imprinting.
set group imprint on
## set body 1 as the only visible entity for better
## visualization of subsequent decomposition
body all visibility off
body 1 visibility on
display
## the geometry must be decomposed into sweepable volumes
webcut body all plane sur10
webcut bod1@A with sheet extended from sur14
## turn the visibility back on
body all vis on
## center of cylinder is at origin.
## Core out the center of the half sphere,
## and include the block imprint inside the core.
## rad 60 was a guess but the user can also use existing curves
## to estimate the radius required to core the center.
## NOTE: “WARNING:  entity ignored ... cutting tool does not”
## will appear.  This is just information to the user that not
## all the bodies were intersected with the cylinder
webcut body all cylinder radius 60 axis y
## now cut in half to rotate the sphere portion. the mesh can
## only be rotated 180 degrees so it has to be cut in half.
webcut body all plane xplane
## now we want 1 volume per body; currently one body has two
## volumes.
separate body all
##turn off graphics to run faster
graphics off
## imprints the profile of mating or contacting surfaces onto
```

```
## each other.compress ids renumbers the entities starting at 1
imprint all
graphics on
## set the element edge size for all the volumes and
## Automatically select the meshing scheme for all volumes
vol all size 15
vol all scheme auto
## After inspection, the 4 volumes that couldn't be meshed, just
## need help with setting the correct surface schemes.
surface 190 198 name 'submap_surfs'
surf 343 348 name 'map_surfs'
submap_surfs submap_surfs@A scheme submap
map_surfs map_surfs@A scheme map
## Try auto scheme again to be sure.
vol all scheme auto
## So one would think we are ready to mesh.
## Lets save where we are.
export acis "advanced_temp.sat"
## But first we need to check to make sure the collection can be
## meshed.
## We need to do a merge all first.
merge all
## Note that no curves or vertices were merged seperatly so
## merging was done successfully.
## Redo the auto scheme because some of the entities were
## deleted in the merge.
vol all scheme default
vol all scheme auto
## Oops. (;-0)Given the other volumes meshing constraints, we
## are left with a collection of volumes that need to be many to
## many sweept(will be in future releases),
## which we need to decompose.
## While only one volume is reported, the other mirroring this
## ones also need to be decomposed.
## Here are the bodies that need further decomp.bod10 andbod108
## Lets reset, to get rid of the merged entities, and read in
## that file that we saved.  Decomposing merged entities would
## cause data base problems...
reset
## Everything is named, and we don't need to keep naming
set default names off
## There will be lots of "Entity name" print statements that you
## can ignore.
import acis "advanced_temp.sat"
## Lets decompose those two bodies.
## bod2 and bod2@A.  the "{Id("group")}" is an APREPRO command
## which is acceptable in CUBIT
## Decompose and group the results into a group.
webcut bod2 plane sur27 group_results
## This command names the group previous created
group {Id("group")} name "right_s"
webcut bod2@A plane sur31 group_results
group {Id("group")} name "left_s"
## Now we still wouldn't be able to mesh.  sweep grouping would
## show a problem. Rotating the mesh around with having
## the many to one sweeps would cause mesh matching problems.
## Esentially with the collection of volumes here we have a
## many to many sweep. So really we need to get each volume
## in the right_s and left_s groups to be 1 to 1 sweeps to get
## the mesh to match correctly.
webcut right_s plane sur25 group_results
webcut left_s plane sur30 group_results
## now chop the top part of both the right_s and left_s sides
## This is further needed to get the group to be sweepable.
## Doing this will generate a separte sweep path.
group "rounds" equals body all in group 4 5
webcut rounds plane sur32 noi nom group_results
## So lets do a final imprint.
graphics off
imprint all
## At this point you could merge all, set a size, do auto scheme
## and group sweep volumes and find out that everything is
```

```
##meshable,with the exception of those surfaces we need to hand
## set the chemes on.
export acis "advanced-decomp.sat"
reset
graphics on
## Now the meshing part.
import acis "advanced-decomp.sat"
merge all
#The 4 surfaces that had names were merged into two surfaces.
group 'maps' equals surface name 'map_surfs'
#oops that got both.
group 'submaps' equals surface name 'submap_surfs'
group 'maps' subtract submaps from maps
surface all in submaps scheme submap
surface all in maps scheme map
vol all size 15
vol all scheme auto
## assign all sweepable surfaces to a group.  This ensures that
## the sweeping will be done in the proper order so the meshes
## will match.
## match intervals ensures that adjacent volumes have consistent
## intervals.
group sweep volumes
match intervals vol all
## First mesh the sweep groups.
```

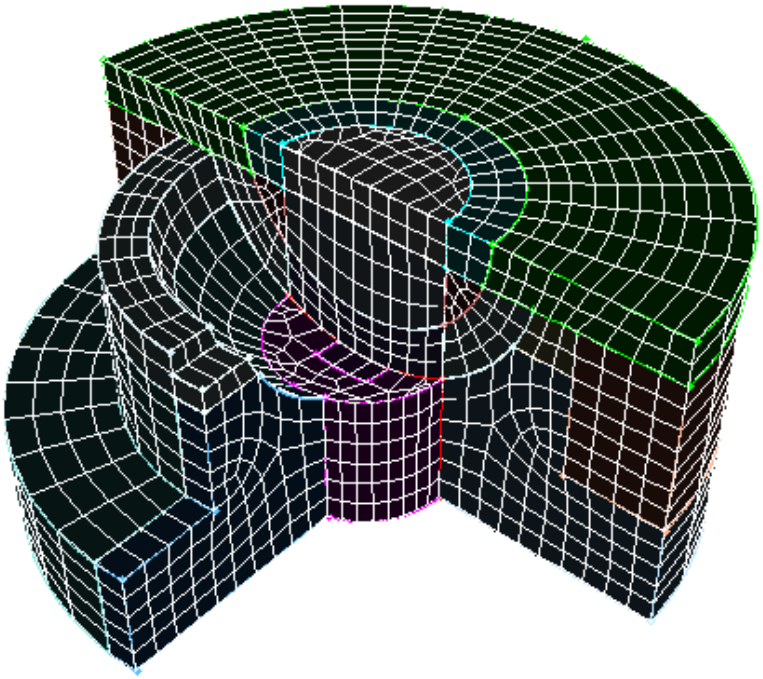


Figure A-7: Mesh of Advanced Tutorial Problem

```
mesh sweep_groups
## Now mesh the volumes that are mappable or submappable.
mesh vol all
#Now fix up the mesh to make it nicer.
delete mesh
#improve the mesh by doing two things:
#1) change the intervals to remove skew
#2) use the boundary adjuster to improve it.
curve 455 int 10
#After changing intervals, you need to resolve the interval
```

```
#solution.
match int vol all
mesh surf 67 77 146 191
#straighten the skew on this surface first
adjust boundary surface 67
#now make the nodes immovable to adjust the mesh on the next
surface
#without this it might undo our last adjustment
surf 67 node position fixed
#now fix the skew on the next surface.
adjust boundary surf 77
#The other two surfaces are a little different.
#The optimal way to releive skew would be to do more
#decomposition.
#We don't want
adjust boundary surf 191
mesh sweep_groups
mesh vol all
## Now lets look at quality.
## The main thing to look for are 1: No negative Jacobains
## 2: No skew near 1.0
quality volume all
## Now take a look at the interior:
body 21 23 25 16 5 3 4 11 27 29 13 15 31 vis off
display
## To show only exterior faces and not geometry do this:
## (this generally gives better pictures)
graphics use facets on
```

▼ ExodusII File Specification

Element Block Definition Examples

Multiple Element Blocks

Multiple element blocks are often used when generating a finite element mesh. For example if the finite element model consists of a block which has a thin shell encasing the volume mesh, the following block commands would be used:

```
Block 100 Volume 1
Block 100 Element Type Hex8
Block 200 Surface 1 To 6
Block 200 Element Type Shell4
Block 200 Attribute 0.01
Mesh Volume 1
Export Genesis 'block.g'
```

This sequence of commands defines two element blocks (100 and 200). Element block 100 is composed of 8-node hexahedral elements and element block 200 is composed of 4-node shell elements on the surface of the block. The "thickness" of the shell elements is 0.01. The finite element code which reads the Genesis file (block.g) would refer to these blocks using the element block IDs 100 and 200. Note that the second line and the fourth line of the example are not required since both commands represent the default element type for the respective element blocks.

Surface Mesh Only

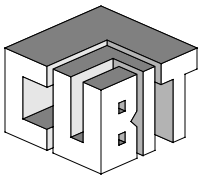
If a mesh containing only the surface of the block is desired, the first two lines of the example would be omitted and the **Mesh Volume 1** line would be changed to, for example, **Mesh Surface 1 To 6.**

Two-Dimensional Mesh

CUBIT also provides the capability of writing two-dimensional Genesis databases similar to FASTQ. The user *must* first assign the appropriate surfaces in the model to an element block. Then a **Quad*** type element may be specified for the element block. For example

```
Block 1 Surface 1 To 4
Block 1 Element Type Quad4
```

In this case, it is important for users to note that a two-dimensional Genesis database will result. In writing a two-dimensional Genesis database, CUBIT *ignores* all z-coordinate data. Therefore, the user must ensure that the Element Block is assigned to a planar surface lying in a plane parallel to the x-y plane. Currently, the **Quad*** element types are the only supported two-dimensional elements. Two-dimensional shell elements will be added in the near future if required.



Appendix B: Available Colors

All color commands in CUBIT require the specification of a color name; Table B-1 lists the colors available in CUBIT at this time. Table B-1 lists the color number (#), color name, and the red, green, and blue components corresponding to each color, for reference.

Table B-1: Available Colors

#	Color Name	Red	Green	Blue
0	black	0.000	0.000	0.000
1	grey	0.500	0.500	0.500
2	green	0.000	1.000	0.000
3	yellow	1.000	1.000	0.000
4	red	1.000	0.000	0.000
5	magenta	1.000	0.000	1.000
6	cyan	0.000	1.000	1.000
7	blue	0.000	0.000	1.000
8	white	1.000	1.000	1.000
9	orange	1.000	0.647	0.000
10	brown	0.647	0.165	0.165
11	gold	1.000	0.843	0.000
12	lightblue	0.678	0.847	0.902
13	lightgreen	0.000	0.800	0.000
14	salmon	0.980	0.502	0.447
15	coral	1.000	0.498	0.314
16	pink	1.000	0.753	0.796
17	purple	0.627	0.125	0.941
18	paleturquoise	0.686	0.933	0.933

Table B-1: Available Colors

#	Color Name	Red	Green	Blue
19	lightsalmon	1.000	0.627	0.478
20	springgreen	0.000	1.000	0.498
21	slateblue	0.416	0.353	0.804
22	sienna	0.627	0.322	0.176
23	seagreen	0.180	0.545	0.341
24	deepskyblue	0.000	0.749	1.000
25	khaki	0.941	0.902	0.549
26	lightskyblue	0.529	0.808	0.980
27	turquoise	0.251	0.878	0.816
28	greenyellow	0.678	1.000	0.184
29	powderblue	0.690	0.878	0.902
30	mediumturquoise	0.282	0.820	0.800
31	skyblue	0.529	0.808	0.922
32	tomato	1.000	0.388	0.278
33	lightcyan	0.878	1.000	1.000
34	dodgerblue	0.118	0.565	1.000
35	aquamarine	0.498	1.000	0.831
36	lightgoldenrodyellow	0.980	0.980	0.824
37	darkgreen	0.000	0.392	0.000

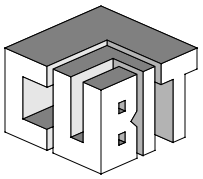
Table B-1: Available Colors

#	Color Name	Red	Green	Blue
38	lightcoral	0.941	0.502	0.502
39	mediumslateblue	0.482	0.408	0.933
40	lightseagreen	0.125	0.698	0.667
41	goldenrod	0.855	0.647	0.125
42	indianred	0.804	0.361	0.361
43	mediumspringgreen	0.000	0.980	0.604
44	darkturquoise	0.000	0.808	0.820
45	yellowgreen	0.604	0.804	0.196
46	chocolate	0.824	0.412	0.118
47	steelblue	0.275	0.510	0.706
48	burlywood	0.871	0.722	0.529
49	hotpink	1.000	0.412	0.706
50	saddlebrown	0.545	0.271	0.075
51	violet	0.933	0.510	0.933
52	tan	0.824	0.706	0.549
53	mediumseagreen	0.235	0.702	0.443
54	thistle	0.847	0.749	0.847
55	palegoldenrod	0.933	0.910	0.667
56	firebrick	0.698	0.133	0.133
57	palegreen	0.596	0.984	0.596
58	lightyellow	1.000	1.000	0.878
59	darksalmon	0.914	0.588	0.478

Table B-1: Available Colors

#	Color Name	Red	Green	Blue
60	orangered	1.000	0.271	0.000
61	palevioletred	0.859	0.439	0.576
62	limegreen	0.196	0.804	0.196
63	mediumblue	0.000	0.000	0.804
64	blueviolet	0.541	0.169	0.886
65	deeppink	1.000	0.078	0.576
66	beige	0.961	0.961	0.863
67	royalblue	0.255	0.412	0.882
68	darkkhaki	0.741	0.718	0.420
69	lawngreen	0.486	0.988	0.000
70	lightgoldenrod	0.933	0.867	0.510
71	plum	0.867	0.627	0.867
72	sandybrown	0.957	0.643	0.376
73	lightslateblue	0.518	0.439	1.000
74	orchid	0.855	0.439	0.839
75	cadetblue	0.373	0.620	0.627
76	peru	0.804	0.522	0.247
77	olivedrab	0.420	0.557	0.137
78	mediumpurple	0.576	0.439	0.859
79	maroon	0.690	0.188	0.376
80	lightpink	1.000	0.714	0.757
81	darkslateblue	0.282	0.239	0.545
82	rosybrown	0.737	0.561	0.561
83	mediumvioletred	0.780	0.082	0.522
84	lightsteelblue	0.690	0.769	0.871
85	mediumaquamarine	0.400	0.804	0.667

CHAPTER



Appendix C: CUBIT Licensing, Distribution and Installation

The CUBIT code is available for use by personnel inside Sandia, any other government laboratory, or to personnel performing work under contract by a US government entity. In addition, CUBIT can be licensed for non-commercial and research use. For more information on licensing of CUBIT, see the CUBIT web page (http://endo.sandia.gov/cubit) or send email to cubit-dev@sandia.gov.

Note: CUBIT installations have use restrictions. THE CUBIT CODE CANNOT BE COPIED TO ANOTHER COMPUTER AND THE NUMBER OF USER SEATS ON EACH COMPUTER OR LAN IS LIMITED. If additional user seats or additional copies of CUBIT are required, you MUST contact us to acquire them.

CUBIT incorporates code modules developed by outside code vendors and licensed to the CUBIT project. Since the number of licenses for these modules is limited, CUBIT cannot be copied and redistributed without notifying the CUBIT team.

CUBIT is distributed in statically linked executable form for each supported platform. Supported platforms include the HP 9000 series running HP-UX[1], Sun SPARCstations running Solaris[2], and the SGI running IRIX[3]. Additional platforms will be added as required; in particular, a port to Windows NT is underway and should be ready shortly.

Instructions for obtaining the CUBIT code will be given after licensing arrangements have been completed.

In addition to the CUBIT executable, the suite of example problems described in this manual is available upon request.

1. HP-UX is a registered trademark of Hewlett-Packard Company.
2. Sun, SunOS, and Solaris are registered trademarks of Sun Microsystems, Inc.
3. IRIX is a registered trademark of Silicon Graphics, Inc.

APPENDIX C CUBIT Licensing, Distribution and Installation

APPENDIX C CUBIT Licensing, Distribution and Installation

APPENDIX C CUBIT Licensing, Distribution and Installation

APPENDIX C CUBIT Licensing, Distribution and Installation

Appendix D: Element Numbering

▼ Introduction

This appendix describes the element node and side numbering conventions used in ExodusII files written by CUBIT. This information is located here for convenience, but is identical to the information presented in [6].

▼ Node Numbering

The node numbering used for the basic elements is shown in Figure D-1. Specific element types of lower order just contain the number of nodes needed for those elements; for example, QUAD4 or QUAD elements use just the first four nodes shown for quadrilaterals in Figure D-1.

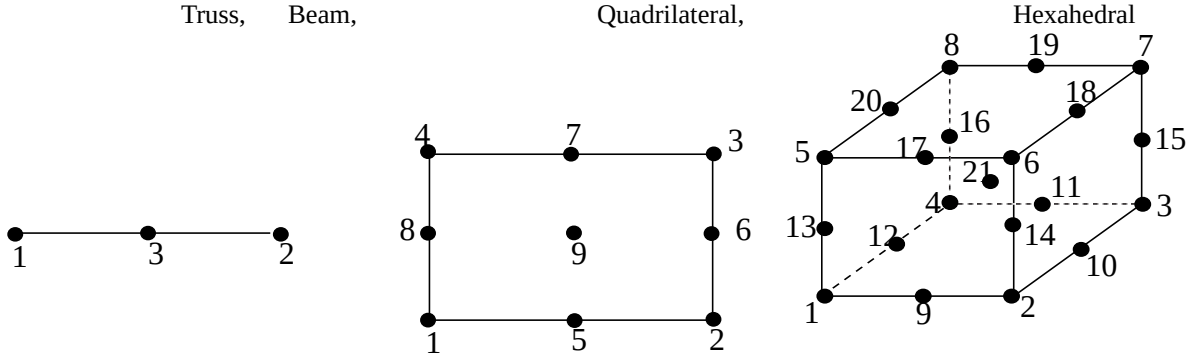


Figure D-1: Local Node Numbering for CUBIT Element Types

▼ Side Numbering

Element sides are used to specify boundary conditions that act over a length or area, for example pressure- or flux-type boundary conditions. Each element side is represented in the ExodusII format by an element number and the local side number for that element. The local side numbering for the basic elements is shown in Figure D-2.

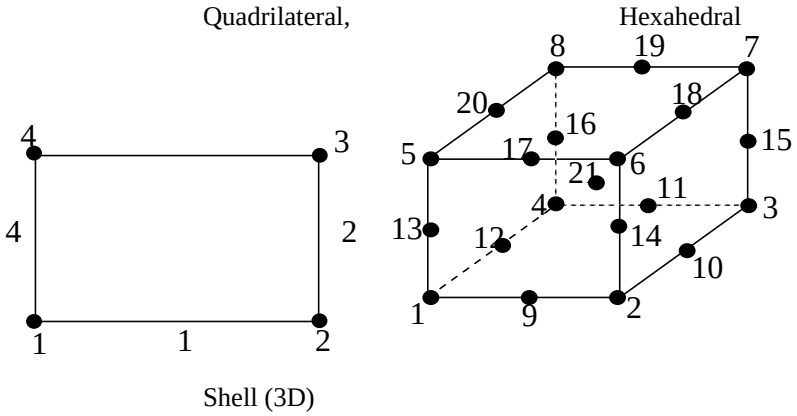
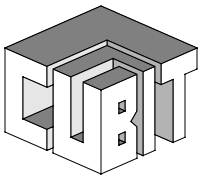


Figure D-2: Local Side Numbering for CUBIT Element Types



Appendix E: Adaptive Meshing

▼ Introduction

CUBIT contains a sophisticated adaptive mesh generation capability for surfaces. This allows the generation of an unstructured quadrilateral surface mesh whose density is controlled by an externally-defined sizing function. This capability has been used to demonstrate adaptive finite element analysis for structural mechanics applications (ref).

APPENDIX E Adaptive Meshing

E

N

S

CHAPTER

Appendix F: Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

MS-0847Distribution:
MS-0321 W. J. Camp, 9200
MS-0841 P. J. Hommert, 9100
9200, All managers (please route to staff)
9100, All managers (please route to staff)
MS-0865 J. L. Moya, 9735
MS-0624 C. A. Neugebauer, 2984
MS-0625 L. K. Grube, 2983
MS-0105 A. J. Webb, 2435
MS-9042 E. P. Chen, 8742
MS-0437 J. Jung, 9135
MS-0828 L. A. Schoof, 9121
MS-0828 J. R. Stewart, 9121
MS-0828 J. A. Schutt, 9121
MS-0828 L. M. Taylor, 9121
MS-0847 S. A. Mitchell, 9226
MS-0847 T. J. Tautges, 9226
MS-0847 D. R. White, 9226
MS-0847 P. Knupp, 9226
MS-0847 R. M. Garcia, 9226
MS-0847 D. J. Melander, 9226
MS-0847 L. Freitag, 9226
MS-0437 S. W. Attaway, 9117
MS-0443 C. M. Stone, 9117
MS-0443 G. D. Sjaardema, 9117
MS-0443 J. G. Arguello, 9117
MS-0834 J. B. Aidun, 9117
MS-0443 M. K. Neilsen, 9117
MS-0443 G. W. Wellman, 9117
MS-0443 J. Holland, 9117
MS-0443 K. Brown, 9117
MS-0443 S. W. Key, 9117
MS-0443 J. D. Gruda, 9117
MS-9042 A. M. Schauer, 8742
MS-0521 S. T. Montgomery, 1567
MS-0660 A. L. Ames, 9622
MS-0826 D. K. Gartling, 9111
MS-0826 M. A. Walker, 9111
MS-0826 R. C. Givler, 9111
MS-0826 P. R. Schunk, 9111
MS-0834 P. L. Hopkins, 9112
MS-0835 R. J. Cochran, 9113
MS-0835 R. R. Lober, 9113
MS-0835 S. E. Gianoulakis, 9113
MS-0557 T. W. Simmermacher, 9119
MS-1109 C. T. Vaughan, 9226
MS-1111 S. Plimpton, 9221
MS-1111 A. Salinger, 9221
MS-1111 S. Hutchinson, 9221
MS-1111 R. Schmidt, 9221
MS-1111 J. Shadid, 9221
MS-1111 D. Barnette, 9221
MS-0819 M. A. Christon, 9231
MS-0819 E. A. Boucheron, 9231
MS-0819 J. R. Weatherby, 9231
MS-0819 S. Petney, 9231
MS-0819 A. C. Robinson, 9231
MS-0820 A. B. Farnsworth, 9232
MS-1166 D. J. Riley, 9352
MS-1186 M. F. Pasik, 9542
MS-0847 CUBIT Report File

Dr. Steve Benzley & Research Assts.
Associate Dean,
General & Honors Education
350 MSRB
Brigham Young University
Provo, UT 84602

Steve Storm
Caterpillar Inc.
Bldg. AD 3335
600 W. Washington Street
East Peoria, Illinois 61630-3335

Dr. Rajit Gadh
347 Mech. Engr. Bldg
1513 University Ave
Madison, WI 53706

Ray Meyers
301 East 925 North
American Fork, UT 84003

Michael Stephenson
2005 West 1550 North
Provo UT, 84604-2212